Universal Integration Architecture for Heterogeneous Datasources and Optimisation Methods

Uniwersalna architektura integracyjna dla heterogenicznych źródeł danych i metod optymalizacji

This dissertation is submitted for the degree of
Doctor of Philosophy

By

Michał Chromiak

FACULTY OF MATHEMATICS, PHYSICS AND COMPUTER SCIENCE,
Maria Curie-Skłodowska University,
Lublin

Advisor:
prof. dr hab. Krzysztof Stencel

Institute of Fundamental Technological Research,
Polish Academy of Sciences

Warsaw 2015

Table of Contents

Page

Listings 5

List of Figures 6

List of Tables 8

Abstract 9

Chapter 1. Introduction 19
- 1.1 Motivation 19
- 1.2 Considerations, Objectives and the Thesis 20
- 1.3 History and Related Work 22
- 1.4 Thesis Outline 23

Chapter 2. The State of the Art and the Related Works 25
- 2.1 Integrity - the Philosophy of Integration 25
- 2.2 Integration - Cure for Chaos of Multiplicity, General Considerations 27
  - 2.2.1 At the beginning there was a relation 28
  - 2.2.2 Revolution - *the Web changes everything* 30
  - 2.2.3 Integration - Principia and Taxonomy 35
  - 2.2.4 Data Integration Practices 38
  - 2.2.5 Integration Theory 42
  - 2.2.6 Data Integration Issues 47
- 2.3 Data Stores - the Integration Targets 51
  - 2.3.1 Database modelling - persistence 51
  - 2.3.2 Relational Model 51
  - 2.3.3 Object-oriented Database Model 55
  - 2.3.4 Column-oriented Relational Database Model (CORDB) – Relational Approach 56
  - 2.3.5 NoSQL – Distributed Storage Services 57
  - 2.3.6 NewSQL 63
  - 2.3.7 Big Data - all or nothing 66
  - 2.3.8 After SQL Era 68
  - 2.3.9 Database taxonomy 70

2.4 Related Works - Overview of Modern Integrating Solutions 71
2.4.1 OLTP & OLAP - sets of operations 72
2.4.2 Metamodels - Metadata 78
2.4.3 Distributed File Systems - Embracing Scaling Up in Size 80
2.4.4 Enterprise Service Bus (ESB) 94
2.4.5 ESB / SOA - Rules of Engagement 96
2.5 Conclusions 97

CHAPTER 3. THE MODEL OF THE ARCHITECTURE 99
3.1 Data vs Application Integration Patterns 100
3.1.1 Patterns in Software Development 100
3.1.2 Architectural Patterns in Integration 101
3.2 General Architecture and Assumptions 103
3.2.1 Virtualization as the Key to Integration – Postulates 103
3.2.2 Polyglot Persistence – building "*The Tower of Babel*" 105
3.2.3 Event Sourcing as a Persistence Technique 108
3.2.4 Command Query Responsibility Separation (CQRS) Pattern 109
3.2.5 OMG CORBA - Standard Specfication 115
3.2.6 Metadata 116
3.2.7 Design Patterns - Study of Utility 116
3.2.8 Integration Database Model - IDBM 118
3.2.9 Indexing Role in Integrated Datamodel 119
3.3 The Architecture 121
3.3.1 *Principia* – Assumptions and Directions 121
3.3.2 Components of the Architecture 124
3.3.3 Workflow 139
3.4 Faced Challenges 141

CHAPTER 4. APPLICATIONS 143
4.1 Integration 143
4.1.1 *Polystores* as the Next-gen Federations vs Qboid-based Architecture for BigData Integration 145
4.2 Optimization 147
4.2.1 Indexing Distributed and Heterogeneous Data 148
4.2.2 Indexing Projections 148
4.2.3 Exploiting Order Dependencies Optimization Technique for Qboid-based Integration Architecture 150
4.2.4 Polyglot Persistence as an Optimization Technique for Integration Architecture 156
4.3 Conclusions 161

CHAPTER 5. SUMMARY AND CONCLUSIONS 163
5.1 The Limitation of Prototype and Further Works 164
5.2 Additional Mediator Functionalities 165

APPENDIX A. PROTOTYPE IMPLEMENTATION 167
A.1 Integration Layer 167
A.1.1 The IDL Scheme for Integration Contexts of Qboid and the Integration View 169
A.1.2 The Integration Scheme in Action – Example 172

APPENDIX B. STANDARDS AND CLASSIFICATIONS 177

APPENDIX C. HADOOP ECOSYSTEM 185

BIBLIOGRAPHY 189

Listings

2.1 OWL/XML Syntax for Ontology Management 41
2.2 GaV on data sources 44
2.3 GaV based query. 45
2.4 GaV query unfolding 45
2.5 LaV S_1_*emp(Name, Age)* 45
2.6 LaV S_2_*emp(Name, Age)* 45
2.7 Declare *emp_type* object with methods - PL/SQL style 55
2.8 Define *emp_type* object with methods - PL/SQL style 55
2.9 Define column and table of *emp_type* type 55
2.10 Query column of *emp_type* type 55
2.11 Column 59
2.12 Super-Column 59
2.13 ColumFamily - simplified notation - i.e. no timestamps and column/super-column names removed 59
2.14 Raw XML based document 61
2.15 JSON-based document; MongoDB style 61
2.16 Metadata document for page node 62
3.1 Employee class. 113
3.2 Employee repository class. 113
3.3 Employee class. 113
3.4 Employee repository class. 113
3.5 Employee repository class – now handles COMMANDS. 114
3.6 Extracted query search handler class. 114
3.7 SQL based FAM selection 126
3.8 *Contributory View* metadata schema. Some parts omitted for readability 126
3.9 Remote Database Object Reference (rDOR) 128
3.10 Contact and Connection Details of a rDOR 131
3.11 Virtual, BRI-based data identification strategy 133
3.12 Exemplary Cell Definition 135
3.13 Exemplary Tuple Definition 135
3.14 Exemplary Record Definition 136
3.15 Exemplary Record Definition 136
3.16 SQL based FAM selection 136
3.17 Qboid Layer 137
3.18 Qboid replica 137

3.19 Qboid replication 138
4.1 BigDWAG selection 146
4.2 Index on Employee's salary 149
4.3 Index on Employee's salary 150
4.4 A query for sales in the indicated period 151
4.5 A rewritten query for sales in the indicated period 152
4.6 Query general schema 152
4.7 PLSQL function that finds minimal Fact_ID for a given date 153
4.8 Simple rewrite with sub-queries 155
4.9 A query for all employees under *Smith* in Oracle's early SQL dialect 158
4.10 A query for all employees under *Smith*, according to SQL:1999 standard 158
A.1 Data hierarchy expressed with XML 167
A.2 XPath expressions definitions 168
A.3 XPath expressions definitions with foreign key 168
A.4 Complete Scheme for Integration Contexts 170
A.5 IntegrationView Explicit Example for 1-101 BRIs 172
B.1 OWL/XML Syntax for Ontology 181

List of Figures

2.1 Increase of web data storage. 30
2.2 Ontology types. 40
2.3 Exemplary local and global (integration schema). 44
2.4 Peer-to-peer data integration 46
2.5 Taxonomy of heterogeneity 48
2.6 Taxonomy of heterogeneity 60
2.7 Traditional OLTP overheads. 65
2.8 Monthly Mobile Data Traffic Forecasts by 2019 67
2.9 BigData: Expanding 3D data space 68
2.10 Database landscape as of 2015 70
2.11 BigData vs Economy [86] 71
2.12 OLTP vs. OLAP collation. 72
2.13 OLAP cube example. 77
2.14 MapReduce concept diagram 81
2.15 MapReduce Shuffle mechanism. 82
2.16 The SQL-like query engines emergence. 85
2.17 Hadoop MapReduce generalization with YARN 88
2.18 YARN-enabled applications running on Hadoop 2. 88

2.19 YARN based application lifecycle. 90
2.20 Stinger evolutionary steps. 90
2.21 Tez API/framework to write native YARN applications 92
2.22 One Tez job instead multiple MapReduce jobs thanks to DAG 92
2.23 *Stinger.next* roadmap. 93
2.24 Compute engines 94
2.25 Service-oriented data distribution. 96

3.1 Polyglot persistence. 105
3.2 Service based polyglot persistence. 106
3.3 Service-based index optimized polyglot persistence. 107
3.4 *Key*, as an index, allows access only to complete record representation. Accessing single attribute/tuple replicas only from within the complete record replicas. 121
3.5 Mediation component layer model. 125
3.6 DRUM – Database Resource Universal Map 129
3.7 *Integration View* context uses Qboid DOR context, however both remain separate contexts. Distinct DORs with id: `001,002,003` describe the same set of attributes while sharing the same BRI 131
3.8 Complete Qboid entity definition covering single and multiple mixed fragmentation patterns. 134
3.9 Slice represents all records that share common global BRI 135
3.10 Metadata driven integration architecture workflow 140
3.11 UML sequence diagram for client request processing life cycle 140

4.1 Complete Qboid entity definition based index 149
4.2 Exemplary database schema 151
4.3 Exemplary table schema with the graph data 158

A.1 Contributory schema mapping 167
A.2 Contributory View of two tables based on XML SAX XPath expressions 169
A.3 *Integration View* of the Employee virtual schema. BRIs 1 to 100. 172
A.4 *Integration View* of the Employee virtual schema. Record 101. 173

B.1 Metamodel standards 178
B.2 Classification of Semantic Heterogeneity Sources (See [203]) 180

C.1 Single User 185
C.2 Multi user. 186
C.3 Query throughput. 186
C.4 CPU 187
C.5 Hive Stinger Evolution 187
C.6 Hive Stinger Phases 188

C.7 Hive versions benchmarks for Stinger Initiative results 188

List of Tables

4.1 Hardware configuration used for tests. 155
4.2 Software used in the testing process. 155
4.3 Test results for 50 request trials. 156
4.4 The hardware configuration used in the experimental evaluation 160
4.5 The software used in the experimental evaluation 160
4.6 The execution times of test queries related to the used data model 161

Abstract

The dilemma of an independent data access is, and always has been, a challenging task. The reader is hereby presented a dissertation that proposes a solution to this problem. This dissertation is focused on transparent and efficient integration of heterogeneous and distributed data, making it available in a unified form to top-level users as a data source origin agnostic repository. Readers should also note that the core of the devised solution is to provide a dedicated architectural model, facing some of the most common issues appearing while the data integration is being considered. Thus the presented solution is capable of interfacing between the legacy data sources (i.e. an arbitrary and already existing data sources) and the virtual, data-agnostic perspective – designed to serve best for client data request calls. What is more, the discussed solution additionally considers enabling fast data retrieval with native methods and auxiliary query evaluation and optimization. The mediating approach for incorporating legacy / integrated data storage engines has provided means to read from integrated sources.

The solution presented in this dissertation, includes a complete approach to transparent data integration. It is based on architecture that utilises Qboid as a main integration entity that plays a main point-of-reference for client data requests. The goal of the architecture is to provide an universal meta-model for accessing heterogeneous (i.e. relational, structural, graph, etc.) data with a unified client request API. The architecture is presented as a middleware that collects information on integrated data sources and thus provides an additional abstraction layer for storing data-model independent optimization techniques for accessing the integrated data. The data access methods are unified regardless of the target data storage engine origin.

The main goal of this dissertation is to design a solution based on the dedicated architecture model that would be able to solve the most profound issues during data integration. Thus the presented solution is able to mediate between the iterated data sources and their virtual- and storage-independent global view. This view will then be used for serving the client data requests. What is more, the proposed solution tends to base its data source access on most basic native and low level access methods available, like the JDBC – that are available for each data storage engine. Secondary goal is to provide a design, that would be elastic and flexible enough to provide means for applying access optimization methods globally i.e. regardless of the local data storage model, or its local optimization capabilities. Such optimization techniques might later be manually or automatically applied for specific and / or defined data request types.

The research on integration of the distributed, heterogeneous, fragmented and redundant data is present since the ’80s of the twentieth century in form of federated databases. However, back then the available set of storing solutions mostly concerned the relational model, less attention was focused on alternative data storage models. Another aspect was that then the data

were in general placed locally and the internet connections did not allow for significant data transfers, not to mention live or streaming communication.
Proliferation of numerous new, dedicated data storage engines and their models in recent years has made the heterogeneity move forward and become a major aspect of data integration. Apart from evolution of relational model into new SQL standards, the model has also evolved and resulted in object-relational, and even purely object data stores. Of course this object-oriented revolution was originating in a great widespread of object-oriented paradigm in programming languages, such as C++ or Java. For past fifteen years or so, also some new data analytical facilities in terms of data warehouses have emerged. The *extract, transform, load* (ETL) environment has emerged and *business intelligence* (BI) has become to play the significant, if not leading role in the data processing enterprise. Dissemination of broadband internet access popularised the online data access across numerous small and medium businesses. The data storage and processing were no more applied only in the domain of scientific researchers and large IT companies. The major boost of electronic data growth, computation and transfer were caused by the online banking, financial operations and trading. Moreover, the projects of digitizing the governmental institutions, their work flow and bringing in the online services for citizens made the electronic data processing as popular as never before in history of mankind. Finally, last but not least, the entertainment area has sealed the online data manipulation to become popular and an important aspect of everyday life in civilized world.

Regardless of the data origin or its goal, there is a serious demand for presenting complete and final business information transparently. This information might be combined of data stored in underlying resources however, must precisely match the client demands and requirements with neither dependency, nor client awareness of the integrated system underneath mechanisms. The data resources, or storage engines in general refer to any data source and / or feed that has become a part of the integrated grid. This includes – the most common – relational databases, object-oriented and object databases, XML / RDF data stores, NoSQL data stores, NewSQL data stores and potentially even the database federations or HDFS based storages. The urging need for integration of such a wide spectrum of different paradigms and models is caused by the public, common, and therefore unspecialised data demand by IT unaware clients.
However, problem is that the front-end software for data access is written in a top-level language that leads to additional complexity while considering low level, query-based, back-end data access. This involves the impedance mismatch issue for the object-to-relational case but this lack of adjacency is present while accessing any type of persistent storage. This always forces some additional data source-specific code to be written.

Therefore the process of integration must therefore, be completely transparent for the end user. Each data request should be independent and unaware of the well encapsulated data source model and structure. This way the mediation – that would be able to conform some common integration and back-end communication schema – must be made available for the top level abstraction. This is done with a mediating approach that is specific for every data source, but on the other hand, assures the common communication scheme for information transfer towards the global, integration abstraction. Thus each mediator must work on a *black box* basis so that the top-level integration abstraction can use it for indirect access to the requested data. This is possible due to the generic nature of the mediator interface for higher levels of abstraction while at the same time the mediator keeps the resource interaction opaque. The global integrator enables access to integrated view with use of some API (e.g. REST) that can be utilized by the client without any prior knowledge of original data characteristics

The top-level integration abstraction works as a virtual metadata repository, storing the data location and access details with the use of a unified schema. All of the local access methods and particularities are supplied with the registered mediators instance of each data source. The resources distribution, fragmentation, replication, redundancy and heterogeneity characteristics are required to be a product of manual configuration, made by the integration administrator

that has to design each and every of the global, integration views.

Keywrods database, data integration, integration middleware, heterogeneous integration, middleware optimization, mediation

Rozszerzone streszczenie

Integracja danych jest problemem rozwiązywanym od wielu dekad, który moim zdaniem nadal nie doczekał się całościowego rozwiązania. Niniejsza rozprawa zawiera propozycje kompleksowego podejścia do problemu przeźroczystego dostępu do heterogenicznych danych. Rozwiązanie oparte jest o architekturę integracji danych wykorzystującą ideę Qboidu. Celem tej architektury jest dostarczenie uniwersalnego meta-modelu integrującego dostęp do heterogenicznych danych (m. in. relacyjnych, XML i NoSQL) w sposób zunifikowany dla zapytań klienckich. Model ten jest zrealizowany w postaci middleware zbierającego informacje o dostępnych źródłach danych jednocześnie pozwalając na złożenie w nim metod optymalizacji dostępu do tych danych. Dostęp do danych dzięki proponowanej architekturze jest realizowany w identyczny sposób bez względu na to z jakiego źródła danych pochodzi żądany zasób informacji. Głównym celem pracy jest zaprojektowanie rozwiązania opartego o dedykowany model architektury, który będzie w stanie rozwiązać najbardziej istotne problemy pojawiające się podczas integracji danych. Dlatego też prezentowane rozwiązanie będzie w stanie pośredniczyć pomiędzy integrowanymi źródłami danych i ich wirtualnym i niezależnym od źródła danych globalnym widokiem. Widok ten będzie następnie wykorzystywany dla obsługi zapytań klienckich. Ponadto, proponowane rozwiązanie stawia sobie za cel umożliwienie dostępu do integrowanych danych za pomocą natywnych metod dostępu, charakterystycznych dla modelu lokalnego źródła danych. Dodatkowym celem jest umożliwienie zastosowania w architekturze integracyjnej mechanizmów pozwalających na stosowanie dodatkowych metod optymalizacji dostępu dla konkretnych typów żądań klientów. Na najniższym poziomie dostępu do każdego ze źródeł danych wykorzystany zostanie dobrze znany wzorzec mediacji. Dzięki temu, możliwe będzie podłączanie się do integrowanych źródeł danych w typowy dla nich sposób. Albowiem inaczej, wygląda wykorzystanie języka wysokiego poziomu do obsługi relacyjnej bazy danych, bazy typu NoSQL czy pliku z danymi. Za sprawą prac prowadzonych nad federacyjnymi bazami danych pierwsze badania nad integracją danych datuje się na lata osiemdziesiąte dwudziestego wieku. Jednak wówczas liczba możliwych modeli przechowywania danych była mała i ograniczała się właściwie w większości do relacyjnych baz danych. Inną cechą danych w tamtym okresie był fakt lokalności danych. Natomiast połączenia internetowe były limitowane nie tylko w przepustowości ale również nie było do nich powszechnego dostępu. Stąd dane miały charakter lokalny a ich przesył był wręcz niewskazany, nie wspominając już o analizie danych przesyłanych na żywo. W ostatnich latach można jednak zaobserwować wzmożony rozkwit nowych źródeł danych o zróżnicowanych zastosowaniach, a więc i modelu. Z tego względu mamy do czynienia ze wzrostem znaczenia heterogeniczności w aspekcie integracji danych. Oprócz rewolucji obiektowej w językach programowania tak jak choćby C++ czy Java, która przeniosła się na nowe modele obiektowo-relacyjne oraz obiektowe, mamy również do czynienia z kompletnie

nowymi paradygmatami. W ciągu ostatnich piętnastu, dziesięciu lat, dodatkowo można było zaobserwować wzrost biznesowego znaczenia narzędzi analitycznych typu hurtowni danych. Ponadto, narzędzia typu ETL czy BI stały się głównym obszarem zainteresowania biznesowego. Nie bez znaczenia dla formy współczesnych źródeł danych i ich wykorzystania jest również rozwój technologii szerokopasmowego dostępu do internetu. Internet stał się nie tylko szybki i bardziej przepustowy, ale dodatkowo dostęp do internetu został upowszechniony w biznesie jak i w codziennym użytku osobistym. Pierwotne składowanie jedynie danych naukowych bądź czysto księgowo-biznesowych okazało się niewystarczające dla współczesnych wymagań. Powszechność bankowości internetowej, transferów finansowych oraz handlu internetowego zmieniła całkowicie ilość i naturę informacji jakie należało utrwalać w składach i bazach danych. Wprowadzenie cyfryzacji w sektorze państwowym również wymogło na urzędach i instytucjach państwowych wprowadzenie dodatkowych źródeł składowania, a elektroniczna droga rozpatrywania spraw obywateli sprawiła, że wykorzystanie źródeł danych stało się tak powszechne jak nigdy wcześniej w historii. Ostatnim, ale nie najmniejszym, czynnikiem różnicowania źródeł danych stała się rozrywka. Cyfrowa rozrywka i sieci społecznościowe spowodowały masowe upowszechnienie się wykorzystania źródeł informacji na wszystkich etapiach rozwoju i życia codziennego każdego człowieka z obszarów rozwiniętych współczesnego świata. Bez względu na pochodzenie danych i ich przeznaczenie, koniecznym staje się ich przeźroczysta prezentacja. Prezentacja informacji może być oparta o dane zebrane z wielu źródeł. Dlatego też częstokroć jest ona poprzedzona zbieraniem i łączeniem danych z wielu źródeł, które następnie są prezentowane jako jeden, wspólny widok informacyjny. Ostatecznie, zwracane informacje nie powinny zawierać nadmiarowych informacji o tym skąd pochodzą ani o tym od czego zależy dostęp. Powinny one natomiast zawierać informacje dokładnie odpowiadające na żądanie klienta. Przez zasoby czy źródła danych rozumieć należy mechanizmy i sposoby składowania danych. Należy tu wymienić takie źródła danych jak choćby – najpopularniejsze – RSZBD, obiektowo-relacyjne bazy danych, obiektowe bazy danych, składy danych XML / RDF, składy NoSQL rozwiązania typu NewSQL, a nawet rozproszone systemy składowania plików typu HDFS czy GFS. Olbrzymie zapotrzebowanie na integrację tak szerokiego spektrum zasobów danych jest spowodowane przez powszechność zastosowania mechanizmów składowania danych w coraz to nowych obszarach pracy i życia codziennego. Wynika stąd jedna bardzo ważna cecha. Chodzi mianowicie o przeźroczystość i łatwość w dostępie do danych bez względu na złożoność ich składowania. Ze względu na powszechność zapotrzebowania na składowanie danych wyspecjalizowana wiedza ekspercka nie może być zatem wymagana od każdej osoby, która potencjalnie będzie chciała wykorzystać dane z wielu źródeł. Odpowiedzią na ten problem staje się właśnie proponowana architektura oparta o Qboid. Zauważyć należy, że oprogramowanie służące jako bezpośrednia droga komunikacji z architekturą integracyjną napisane jest z wykorzystaniem języków programowania wysokiego rzędu. Zapewnia to dodatkowe ułatwienia dla programistów w dostępie do żądanych danych, do których dostęp w przeciwnym razie musiałby być niskopoziomowy i oparty o zapytania. To właśnie tu pojawia się zagadnienie obiektowo-relacyjnego niedopasowania impedancji. W takich sytuacjach zawsze wymagana jest dodatkowa złożoność kodu, czy to postaci gotowych frameworków typu ORM, czy własnego kodu wykorzystującego sterowniki typu JDBC.

Nie do przecenienia jest fakt, że dobrze zaprojektowana integracja danych musi być przeźroczysta dla końcowego użytkownika. Każde żądanie skierowane do integratora musi być niezależne i niezwiązane ze sposobem pozyskiwania danych z różnych źródeł. Z pomocą przychodzi w tej sytuacji podejście oparte o mediację. Dzięki niemu, każde źródło danych jest obsługiwane przez dedykowany kod wyspecjalizowanego mediatora. Z drugiej strony, każdy mediator zna wspólny schemat komunikacji pomiędzy każdym mediatorem a integratorem. W taki sposób dostęp do lokalnych źródeł jest zapewniony, przy czym dodatkowo mediator jest w stanie przekazać informacje z lokalnego źródła do globalnego integratora. Integrator z kolei, już w transparentny sposób, przekazuje te informacje w odpowiedzi na żądania klientów. Każdy

mediator wobec tego, pracuje w sposób niewidoczny dla klienta, przy czym integrator posiada jedynie dane, które mediator udostępni dzięki wspólnemu schematowi komunikacji. Sam integrator udostępnia widok danych dzięki prostemu API (np. REST), dzięki któremu dowolny klient może wykonać zapytania bez uprzedniej wiedzy na temat charakterystyki danych. Główny integrator jest swoistego rodzaju abstrakcją działającą jak wirtualne repozytorium metadanych opisujących docelowe dane. Instancja integratora w postaci Qboidu przechowuje wszystkie dane adresowe i dostępowe dzięki uniwersalnemu schematowi. Wszystkie lokalne metody dostępowe i charakterystyki danych są dostarczane dzięki zarejestrowanym w Qboidzie mediatorom poszczególnych źródeł danych. Główne problemy wynikające z integracji danych takie jak ich rozproszenie, różne schematy fragmentacji, nadmiarowość czy heterogeniczność ich źródeł są reprezentowane przez ręczną konfigurację wykonywaną przez administratora integratora, który jest odpowiedzialny za zaprojektowanie każdego widoku integracyjnego.

Acknowledgements

First and foremost I want to thank my advisor professor Krzysztof Stencel. It has been an honour to be his Ph.D. student. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. As a great person Krzysztof has always let me know that the problem is to be solved in a "look how simple it is" way. The accuracy of stating goals and clear justification has made our hard work to become purely an intellectual game with lots of fun. It was also an absolute privilege to meet Krzysztof not only as a successful young Professor with distinguished career, but also as a great father's role model, with beautiful and loving family.

I will forever be thankful to my former college research advisor, Piotr Wiśniewski. Piotr has been helpful in providing advice many times during my research. He has always had a good tips and ideas for significant research topics, which – thanks to his expertise – always somehow tend to meet the assumed goals. The ease of finding correlations for future research and the unquestionably reasonable and solid methodology approach – that along many discussion were indisputable – are the virtues that I will always admire.

I also thank to Professor Kazimierz Subieta for a countless seminars that has gathered many great, ambitious and beautiful minds. Among them was my future advisor Krzysztof. Those seminars has introduced me into the world of scientific level of researches that had been widely discussed across most significant standardization organizations and competed with world's biggest enterprise companies. Those meetings made me to pursue a career in research.

Chciałbym przede wszystkim podziękować mojej rodzinie za wsparcie i wiarę w sukces. Mojemu Dziadkowi Gabrielowi zawsze będę wdzięczny za to, że mimo problemów zdrowotnych, był moim kompasem i nauczył mnie jak w życiu znajdować właściwy kierunek, oraz wyznaczać porządek i ciężko pracować. Dzięki niemu poznałem czym jest nauka i czym jest odpowiedzialność i dojrzałość. Moim rodzicom i babci, który wychowali mnie w wierze i pasji do nauki od najmłodszych lat oraz zawsze mnie skutecznie wspierali w najtrudniejszych momentach. Mojemu bratu Jakubowi dziękuję, za to że dzięki niemu powstała ta właściwa i rozsądna część pomysłów, oraz za to, że miał dla mnie zawsze odpowiednie słowo.

Najmocniej wreszcie dziękuję mojej kochanej, wspierającej, mobilizującej i cierpliwej żonie Asi, której wierne wsparcie podczas końcowego etapu prac było najlepszym wzmocnieniem. Mojej córeczce, Gabrysi dziękuję najmocniej za to, że pojawiając się sprawiła że tata dostał olbrzymi zapas sił, motywacji i radości do dalszej pracy.

Dziękuję Wam wszystkim.
Michał Piotr Chromiak
Lublin, 3rd August, 2015

CHAPTER

1 Introduction

"There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things."

— Niccolo Machiavelli, *The Prince (1532)*

In modern world the amounts of stored computer data – as a form of information – has been rapidly growing since the beginning of the Internet era. This data, often scattered across the Internet, suffers from all of the issues known to the data integration domain. It includes distribution, fragmentation, redundancy (i.e. replication) and the most importantly – heterogeneity. Hence, gathering such information and presenting it in the human understandable and readable form requires much effort. Getting the requested aspects mostly involves human interactions that requires laborious and error-prone analysis.

At present human part is still crucial to the integration process, its influence should be reduced to minimum and made self-checking.

1.1 Motivation

The integration of data sources, as the data storage facilities, in modern world mainly focus around the databases in terms of DBMS. A data source can be considered as every piece of software or hardware that can be accessed for requested data. This includes file systems, spreadsheet programs, networking services or memory access. Hence, the digitized data can be recorded as a stream, or a piece of memory in multiple ways. Considering the number of data models and their related access methods we have a serious problem of abundance. While it is not bad per se, but when a considerable unified interface is the goal – the solution would require an accurate and holistic understanding of the problem. Prudential assumptions for thoughtfully designed architecture requires comprehensive justification of design choices and decisions. Thus, a firm theory of software engineering would be taken into consideration during the design, implementation and development of postulated architecture.

Along the years software engineering has defined some common situations where a pattern can be found. These frequently occurring circumstances have also been classified and some common solutions have been proposed [1]. Based on the accurate understanding of software engineering conceptual tools, the general integration interface requires a deeper discussion on variety of its aspects.

The goal of this dissertation is to design and implement an initial base for a major architecture capable of providing a unified data access interface. As the integrated data will originate

from many – presumably enterprise-sized – data sources, the proposed solution must be designed to be lightweight. While combining data from multiple data sources, potentially with high volume, this is a prerequisite not to move the actual, stored data from the original location. This condition applies to the integration process. In other words the integration should not collect the data itself. On the other hand, the developed architecture should provide special means for its clients, to receive the target data located at the integrated data source.

1.2 Considerations, Objectives and the Thesis

The idea designed and implemented in this thesis, assumes a virtual integration of an arbitrary data source into a form of a central and homogeneous, non-redundant, schema independent and consistent virtual meta-repository. Such unified data access interface requires considering safe (i.e. assured reliable results) and secure (i.e. authorisation, privacy, roles etc.) infrastructure. The proposed solution is designed to work with any kind of registered data source and can be easily extended to work with new data sources, not even considered as of the moment of initial development. All of the dissonance between the data models (e.g. the impedance mismatch) and the high-level programming languages used by developers, will be resolved due to dedicated architectural design of the proposed solution. It is also not likely that each programmer would learn / know every data source's particularities.

These factors reveal a justified need for providing a transparent and effort-less API to the legacy data sources, enabling effective and straightforward data manipulation without significant additional resources, by simply using lightweight and high-level programmer friendly REST API.

The novel approach of this dissertation involves transparent integration of heterogeneous, fragmented, replicated and distributed data sources utilizing dedicated API. This API is available to client's requests and supplies a common data integration perspective/schema consisting of data access objects (a.k.a database object reference – DOR) enabling fast and native access to actual target data. The native access methods are crucial to get the highest speeds while accessing and manipulating data at the legacy data source. The client is not aware of actual resource location, data model or even role which is used while connecting. The legacy data source content is mapped to specially designed integration schema that can be created regardless of the local schema at the data source (if any). Therefore integration schema can be transparently queried by client's calls using the API, for integrated data that is being described by this schema. Due to the requests via Representational State Transfer (REST) – which is a software architecture style based on HTTP stateless protocol – communication is narrowed to well known, basic concepts available and easily mappable in every major high-level programming language that client might use.

However, this versatility must involve a kind of a translating layer that would enable covering all of the integrated data sources' particularities. Such a dedicated interface would be capable of data communication between the particular data source and the integrating instance. This piece of software will be referred to as a data source *mediator*. The mediator has two types of interface – a *reporting interface* and the *data source interface*; however, the latter one – the *mediator-to-data source* communication interface, can only be used by the mediator itself. In other words – the mediator has to work as a black box so that the wrapped resource would be *opaque* to the upper layers of the integration architecture. The last assumption about the mediator is that its reporting interface is generic; meaning its reliability and functions are independent of the data source type.

The important part of the integrating architecture assumptions is its *transparency*. Each client's request is made towards the integrating service API and is not aware of the actual data source particularities, like, model, location, schema, role etc.

As the integrated data potentially involve enterprise-sized data sources, *materialization* of the

integrated data apart from its origin location, is not considered. This does not include cases when the requested results must somehow be returned to the calling client.
Additionally, considering potential integration of replicated data sources, the integration schema would have to consider *replication* representation by design, due to potential optimization gains regarding current particular data source workload and networking performance.

The concluding theses are:

1. **Legacy data sources can be transparently integrated and interfaced with a Qboid based virtual meta-repository, while the data could still be gathered utilizing RESTfull service, without additional data manipulation, nor replication.**

2. **Well known optimization mechanisms and dynamic performance metrics can be implemented for Qboid based architecture, enabling optimized data access without interfering with the local data source potential optimization engine, or its data schema.**

The thesis has been verified and tested accordingly with the use of prototype implementation based on software engendering S.O.L.I.D. principles and modular approach. This is not only imposed by the *clean code* postulates but also required due to highly complex nature of the problem.

At the beginning the state of art in the field of numerous data storage engines and present integration solutions for those heterogeneous data sources were analysed. The most notable set of solutions has been isolated with their assumptions, advantages and weaknesses, considering their purpose and possible adaptation for the introduced architecture – *Chapter 2*. The resulting conclusions from this work, combined with the tenet of virtual metadata repository and the author's experience based on commercial software craftsmanship best principles and practices [1] , enabled a design of an Qboid-based architecture that tends to face the most prominent integration issues – *Chapter 3*. The goal for the devised architecture is to become a lightweight and generic solution – in terms of communication and data transfer – that provides a flexible and reliable platform for integrating every data source equipped with a dedicated mediator component. Due to prototypical, read-only nature of the presented solution a dedicated approach based on CQRS is postulated. Based on CQRS the author additionally propose a couple of approaches that might be considered when balancing between the separation and complexity while reducing architecture's coupling – *Chapter 3.2.4*.

For the purpose of client access optimization, advanced design patterns have been considered for utilization, in order to allow effective client handling without the need of platform-dependent multi-threading – *Chapter 3.2.7*. The same Chapter also considers the architecture's resemblance to the well defined, and accustomed standards such as OMG CORBA – *Chapter 3.2.5*.

The first prototype implementation lacks functionalities of a complete enterprise class solution. However, this allows transparent and reliable access to the integration-participating resources for the purpose of testing its utility value towards application of source-independent optimization techniques.

This stage of analysing various optimization techniques – *Chapter 4* – involved the usage of three well documented and tested optimization approaches. The goal was to prove that the architecture is flexible enough for arbitrary optimization. It involved the process of accommodating the architecture with optimizations based on different approaches, varying from well known indexing, through order dependencies – involving query rewriting (*Chapter 4.2.3*), to the advanced, task-oriented data storage model appliances according to the data's nature – *Chapter 4.2.4*.

[1] Such as agile development (SOLID, DRY, KISS), clean code, low coupling recommendation, and test driven aspects.

Devising of the architecture was based on the analysis of existing solutions that were considered related in terms of data integration architecture. The conclusions resulted in some general assumptions and considerations that become the basis of the proposed solution. The postulated architecture assures completely transparent read access for arbitrary, registered data which is presented by Qboid's integration view.

Apart from access transparency, the most effort has been devoted to efficient data transfers. This feature involves using pure metadata for describing the target *"heavy"* data. With such an approach, the actual data is being transferred only on *as-needed*, lazy basis – while all of the optimization, localization and access have been assured by the metadata.

The prototype testing environment realising the above goals and functionalities has been implemented with Java language. Additionally, for the sake of future development existing Spring Framework and JDBC drivers have been used for connecting and developing the client REST API. The universal approach of dependency injection has also been used across the entire prototypical testing environment. Networking connections are based on the TCP/IP stack.

1.3 History and Related Work

All kinds of digital video, audio, statistical, financial and industrial data whose volumes are kept growing almost exponentially – forced multiple dedicated data storage technologies. This in turn, caused a need for data integration. However, the integration is not a new area of interest and research. The aspects of data integration have already been considered in 1980's while the concept of federated databases first emerged. Next was, the introduction of mediator and wrapper concepts formulated by G. Wiederhold in [2] in early 1990's .Since then there were many solutions based on the mediation model in different ways and flavours [2] . The mediator was defined as:

DEFINITION 1.1: *Mediator*

Mediator – an autonomous software module capable of processing data from an underlying data source, according to the general system requirements.

The mediators are expected to be resource independent. The second concept considered in [2] is a *wrapper*. It is considered as a piece of software that lies between the mediator and the target data source. What is more (just as in the case of the presented solution), the Wiederhold states [2, p.17] that there is no need for a *user-friendly* mediator's interface. What mediator's interface is expected to be is the machine- and communication-friendly interface. This is the responsibility of the client top-level API and client application to provide applicable data display depending on the client type. The goal of the actual mediation process is gathering the information about data stored at storage engines. This should be executed in the user-undisclosed manner – while transparently, efficiently and reliably delivered. The approach presented in [2] has been followed by numerous implementation from early projects such as federated multidatabase Pegasus [3–5], Amos [6], heterogeneous distributed database DISCO [7], to more modern and rather enterprise related: *IBM Infosphere Information Server for Data Integration* [8], *Oracle Data Integrator* [9] (oracle-oriented), Oracle GoldenGate [10] (heterogeneous), *Pentaho* [11], *Talend Data Integration* [12].

Next in evolution there were various *Object-Relational Mappers* (ORMs) and *Data Access Object* (DAO) solutions aiming at mapping the existing data from the relational systems into object-oriented structures of programming languages. On the other hand, these solutions also

[2] None were closely related to the number of issues considered by the proposed solution, which actually also uses mediators as low-level layer data source connectors.

tended to provide means for persisting programming language objects. It was also the time when the first object-oriented databases were introduced as an alternative to those based on relational theory [13].

As of the federation model, it has evolved together with the mediator-based concepts in 1980s and early 1990s. The representatives of that era, were early prototypes of mediators used in the federated systems like: *TSIMMIS* [14], *Garlic* [15], and *Mariposa* [16]. The research also involved federated data based on disparate schemas [17] (later elaborated also in [18]), and application semantic mismatch resolving [19]. Alike the presented architecture some research was postulated back in 2005 [20] to use a concept of a general *dataspace* that should be able to model any kind of relationship between two (or more) participants. This idea can be interpreted in terms of Qboid's virtual data integration views.

The most recent research, like the BigDWAG project, is also based on the mediators used in the federated database approach. In contrast to previous understanding of mediator as an instance that is expected to cover domain-specific functionality, BigDWAG use mediators (i.e. BigDWAG *islands*) to span multiple data models. On the other hand, the same as the Qboid proposed in this dissertation, BigDWAG also aims to focus on data replication and its location – unlike the generic mediator concept. It is worth mentioning that while the Qboid based architecture can easily provide query execution across multiple data storage engines (see Chapter 4.2.4) with result joining, and schema ready for metrics, whereas BigDWAG still lacks [21] partial query executions and adaptive query processing depending on the data source performance status. However, it must be noted that Qboid architecture is a very basic prototype whereas BigDWAG is a fully functional prototype that has been developed by the experienced and dedicated team.

Currently the most widespread solutions for data integration are the extract, transform, load (ETL) systems that can build schema, handle error cleaning, provide attribute transformation rules to common units, and remove duplicates. However, in terms of modern days data nature this solution has become very costly. The bulk-loading data warehouses that used to keep all of the data, presently tend to fail while considering the real-time data streams, and additionally mostly not ready for semi-structured or text data. The ad-hoc solution to these problems is provided in the form of *data lakes* where all data are placed. In such case the persistence is handled by file systems that mostly base on open-source HDFS, and Hadoop solutions. The down side of this solution is that still it lacks important features like true transactions and its batch nature makes it not suitable for all kinds of applications. However, this began to change during last year [22].

Due to enormous research attempts last two years, it is almost certain that in the nearest future the data storage mechanisms will evolve significantly compared to what legacy and partial / ad-hoc solutions are presently available.

1.4 Thesis Outline

The thesis is divided into the following chapters:

Chapter I: Introduction The first chapter presents the motivation as well as historical and present research landscape of the exploration of the mediator-based heterogeneous data integration. It depicts the dissertation objectives and the technologies considered briefly.

Chapter II: The State of the Art and Related Works The state of the art and the related works chapter conducts the discussion of data source model taxonomy and the reasons for present data storage model types proliferation. It additionally focuses on theoretical approaches that have already been considered in the literature. An extended view is being considered regarding distributed storage solutions as one of the prospective future evolutionary steps for storing data

Chapter III: The Model of the Architecture This chapter considers the patterns and general assumptions that should be taken under account while the Qboid based architecture is involved. It also compares the Qboid based architecture to the concept of OMG CORBA Implementation Repository that the Qboid is partially patterned on. The chapter contains an in-depth discussion of the Qboid architectural layered design. It also elaborates on the communication schema that has been designed for the architecture, and a detailed description of its components. The chapter also describes and follows a complete workflow of the Qboid-based architectural life cycle.

Chapter IV: The Applications This chapter discusses the proof-of-concept testing optimization scenarios of the postulated Qboid integration architecture prototype. It provides detailed experiment assumptions and testing environment descriptions. The tests were based on relatively large resulting sample data for heterogeneous data storage engines. Additionally the chapter includes the tests results confirming the expected utility of the Qboid-based architecture for the integration and heterogeneous optimization purposes.

Chapter 2

The State of the Art and the Related Works

"Integrity is a concept of consistency of actions, values, methods, measures, principles, expectations and outcomes."

— *The Definition*

As a Ph.D. dissertation it deserves to start with some philosophical justification for the discussed solution due to settling the taxonomy and understanding how definitions have been recognized in terms of proposed approach.
The following sections will also further process to some more in-depth overview of existing solutions. However, one has to be warned that, considering the area of data storage and DBMS it must be noted upfront, that the statements gathered here as an overview are accurate as of the time of writing this dissertation. This is due to multidimensional, rapid changes in this field. Thus, a level of abstraction has been preserved to provide more time-proof knowledge. For the same reason, more detailed descriptions of chosen solutions have been carefully specified with version classification for reasoning precision.

2.1 Integrity - the Philosophy of Integration

Consistency. Consistency is the key word that defines the way to approach final understanding of the data integration problem. Presently our civilization is witnessing an informational revolution, just as it was in the 18th century with the steam and combustion engine during the industrial revolution and the electrical revolution at the beginning of the 19th century. Just as it happened in the past, with steam and electricity - when everything changed to adopt the newly discovered phenomena - presently our reality is being *digitized*. This means that in the evolutionary order of things, we try to map the existing natural (i.e. as we percept it) *information* dialect - reality - into newly discovered conditions i.e. the computer based inscription.

To succeed with this mapping, some definitions need to be formulated. First of all, what is *information* - as the mapped subject - then? For the purpose of this dissertation, let us say that information is a measure of entropy. On the other hand, we have to follow the definition of an entropy as a measure of *unpredictability/uncertainty* of *information content*.

Definition 2.1: ***Entropy***

In terms of Information Theory *Entropy* is the average amount of information contained in each message received.

In other words, the more information, the less chaotic environment we deal with. Now even though sometimes we can not explain some of the phenomena - this does not mean that they

are chaotic, we simply lack detailed information. So if the probability of an event A is 1 then its entropy is 0.

$$\Pr(A) = 1 \Longrightarrow H = 0, \qquad H \in [0,1]$$

On the other hand the entropy of e.g. coin toss, is as high as it could be, only when the probability of heads is the same as the probability of tails.

$$\Pr(Heads) = \Pr(Tails) \Longrightarrow H = 1, \qquad H \in [0,1]$$

Now if we recall the physical research on micro-structure of the universe, we can find out that the real fabric-made particles, regardless of the model, are located in great distances from each other. Therefore we can state that every aspect of reality and thus - our life, consists entirely of information defining an extremely sparse matrix of physical fabric.

Now let us refer to the digitalization of the information mentioned earlier. In computer world this form of information is refereed to as the *data*.
To effectively handle information – or its digitized form – we always need some schemas to be settled. In real life the schemas are passed to us as a raising process or discovered autonomically as a result of learning. From the computer science point of view, an *ontology* as a formal and explicit specification of common conceptualization is required. However, respective computer model of data acquisition and ontology learning in the area of databases is not as dynamic as the real life learning. Therefore the ontologies and schemas must be settled explicitly and without any ambiguousness. Nevertheless some fields of computer science like artificial intelligence or semantic web try to use RDF or XML languages to organize data; however, it is done only for specialized and limited complexity ontologies. Despite such attempts to mimic the reality, learning, intelligence, and the information storing capabilities by modern computer systems and algorithms, they are still not even congruent to natural effectiveness of human brain.

Prior to remembering information or primitive data storing a way of the data acquisition needs to be developed. Therefore, the next step in evolution of information utilization is the *communication*. In society or adequately in computer world networking – the environment, communicating and sharing data become gradually even more complicated, depending on the number and diversity of communicating sides. The communication means that the stored data has to be transferred between storing sides while keeping (over entire communication life-cycle) its accuracy and consistency - namely the *data integrity*. Data integrity in general considers two aspects. The basic concern is to assure physical integrity by correctly fetching and storing the data. This includes using redundant hardware, error correction codes (ECC), uninterruptible power supply (UPS), some of the RAID arrays, hash function, file systems with block level checksums etc. All to prevent some unexpected external challenges. However, such techniques must be used in considerable combinations. For instance, in the field of databases, assuring that we use reliable transactions [1] is not enough because the RAID controller or the hard drive's internal write-cache might not be accustomed for this purpose. The second step is assuring the correctness of the data in a concrete context, i.e. *logical integrity*. Logical integrity assures that the data simply makes sense under defined circumstances. To ensure this kind of integrity, databases e.g. include mechanisms such as *check* and *foreign key* integrity constraints that makes the stored data fit into the conceptual assumptions of fixed schema. Both logical and physical integrity can be disrupted by same factors like human error, design flaw, concurrent requests for record manipulation. In such a case those factors violate one of the integrity constraints that originates in the relational model but might partly be translated to the concepts present in other models.

Let us then consider some of the most significant integrity constraints.

[1] In the context of databases, a single logical operation on the data is called a transaction. A transaction is reliable when it is ACID (Atomicity, Consistency, Isolation, Durability) - see [23]

- The most general is the *entity integrity* which must contain the *best row id* that can have multiple forms regarding different models: primary key (relational), object id (object), key (key-value stores), document id (document stores) etc. Typically enforced through indexes or constraints like *UNIQUE* or *PRIMARY KEY*.

- Things get more complicated when we consider the second constraint i.e. the *referential integrity*. Referential integrity ensures the relationships between tables remain preserved as data is inserted, deleted, and modified. It is based on the assumption that a foreign key in one table refers to the link field (mostly primary key) in another table. In other words it should not be possible to remove the linked record while leaving the referring record, or throwing an error. In simple terms, the referential integrity guarantees that the target reference ('refers' to), will be found. While with RDBMS it is pretty straightforward because we consider a foreign key to primary key relation, it is not obvious to fill the referential integrity constraints by other models. The NoSQL databases in general were designed to scale (See Definition 2.2 p. 31) better than RDBMSs. However, this has been achieved at the cost of consistency, of which referential integrity is part of. Thus, most of the NoSQL solutions do not support the referential integrity and it becomes an application responsibility to assure such a constraint. There is, however, one exception to this rule i.e. the graph databases [2], that were designed especially to support explicit relations in the form of edges between nodes. So, if a node is deleted, its edges will be deleted too, preserving the constraint [3].
  In general this constraint is enforced through the *FOREIGN KEY* and *CHECK* constraints.

- *Domain integrity* requires more effort to define it properly. Domain integrity validates data for a column of the table. It requires that the primary units of data are atomic. Domain integrity ensures the data values inside a database follow defined rules for data type, length, size, values, range, and format. They are typically enforced through the following constraints: *FOREIGN KEY*, *CHECK*, *UNIQUE*, and *DEFAULT*.

- *User-defined integrity* is the most general and arbitrary of the integrity categories, and it enables setting up some business rules, defined by the user that do not fit the remaining categories. However, all of the integrity categories support the user-defined integrity.

Since 2012 most of the modern databases support integrity constraints and *consistency model* (i.e. predictable results) of data storage and retrieval.

Apart from the discussed mandatory integrity requirements, we should also consider *data retention* policies – based on the political or privacy protection ground – that ensure adherence to the laws and regulations concerning specific data and rules defining the period of time it can be retained in a particular database.

2.2 Integration - Cure for Chaos of Multiplicity, General Considerations

The problem of multiple data sources has emerged together with the immense growth of computer utilization in almost every aspect of everyday life and widespread of broadband Internet access. Due to the enormous number of data being generated in distinct ways, the need to consolidate such data in a unified way has become an urgent demand of modern computer world. The solutions like federated databases or the mediator based architectures emanate this pursuit of data access unification.

[2] E.g. Neo4j which actually supports not only ACID transactions but also guarantees that there are no dangling relationships.

[3] Still, for some cases, like where data-model requiring dynamic, post-priori schema, with meta-schema elements, it would not be a handy choice to use RDBMS, even with its referential integrity.

To understand the complexity of integration process one has to be aware of the past and current achievements in the field of data integration. To obtain the expected goal, which is the access unification, it is required to gain some kind of translator or mapper that could expose the intended type of stored data in the form of canonical protocol. Rapid database adaptation, started [24] in 1960s [4] , led in a natural way to the need of integrating and merging the proliferating types of data repositories. Therefore, as of the 80s of the last century, the concept of federated databases has been risen. This was a response to a need of integrating the databases while covering distribution, heterogeneity, data fragmentation and redundancy. Moreover, since late 80s of the twentieth century there have been numerous papers on the wrappers, mainly focusing on relational databases. Motivation behind the emerging wrapper technologies was to diminish the difference in approach between the legacy data source (e.g. database) content manipulation protocols, and the superior data request calls that should not or can not be aware of the underlying database undergoing mechanisms.

2.2.1 At the beginning there was a relation

The leading solutions since the beginning of the database era have been based on the relational model proposed by Edgar F. Codd in [25]. Codd proposed change of *CODASYL* [5] (network model) approach from the linked list of free-form records into the table of fixed-length records while each table represented a different type of entity. The linked list model was inefficient while considering sparse cases when any part of the data record could be left empty. Codd also introduced the data normalization [6] (see also 2.3.1) concept that assumed refractorization of data into smaller tables [7] according to its schema. This data isolation caused its manipulation to deal only with the adequate table and propagate through the rest of the database using the defined foreign keys. Compared to the previous concepts of CODASYL, no pointers and links rewriting was necessary when the database content was to evolve. The relations of one-to-many (hierarchical model) or many-to-many (navigational model) allowed to reference between the entities and thus, enabling efficient data modelling. The relational model occurred to be stable and well-suited for the client-server programming. It become a predominant technology used for storing schema based data for web and business applications. The fundamentals of current RDBMS were defined eventually in [26] and [27] (System-R [28, 29]). Additionally, in order for DBMS to operate efficiently and accurately, the general assumption for the relational model was based on the *ACID* rules for the transactions [8] to follow. The rules were introduced in [30] and finally the ACID term was coined in [23].

[4] The term "database" was first coined in 1964 by the military information system workers with reference to the collections of data shared by end-users of time-sharing computer systems. At that time the civil terminology was rather "integrated data processing" but soon evolved to utilize the "database" term with reference to data collections that fit the consolidated data requirements.

[5] A consortium formed in 1959 to guide the development of a standard programming language that could be used on many computers.

[6] The process of organizing the attributes and tables of a relational database to minimize data redundancy.

[7] Table (set of rows) can be considered a convenient representation of a relation (set of tuples), but the two are not strictly equivalent. For instance, the relation with three attributes and five values can be represented as a table with three columns and five rows. The table may have duplicated rows (values) while the relation can not have duplicate tuples.

[8] Transaction – a single logical unit of work performed against the database. Transactional operations are coherent and reliable, whereas independent of other transactions.

INFOBOX 1: *ACID priciples*

- **Atomic** : A transaction is a logical unit of work which must be either completed with all of its data modifications, or none of them is performed.
- **Consistent** : At the end of the transaction, all data must be left in a consistent state.
- **Isolated** : Modifications of data performed by a transaction must be independent of another transaction. Unless this happens, the outcome of a transaction may be erroneous.
- **Durable** : When the transaction is completed, effects of the modifications performed by the transaction must be permanent in the system.

While Codd developed the structure for relational databases, the declarative query language was also developed by the IBM employees Donald D. Chamberlin and Raymond F. Boyce [31]. The query language evolved from the terse mathematical syntax of *SQUARE* [9] into *SEQUEL*, with the linear notation and block-structured English keywords syntax easy to modify and maintain; and finally due to the trademark collision with other companies the name was changed by removing vowels to form *SQL*.

Due to constant evolution of modern technologies, the distance between the relational databases (design originated in 60s) and the current, leading methodologies (modelled with complex UML) has increased. This includes the software engineering evolution towards the object-oriented paradigms of modern programming languages (e.g. Java, C++, C# etc.) and even more interesting, new middleware designed for Internet communication like the CORBA standard or the XML/RDF based web technologies. The evolution also brought some novelty to databases.

Considering emerging object-oriented programming in 1980s, the designers and programmers also moved from treating attributes of table records as extraneous and individual fields to the more object based approach. This switched the main perspective from the database relations scope to the relations between the objects and their attributes. That is when the object-relational *impedance mismatch* problem came in first. The problem can be described in general, as a set of difficulties while migrating the data from the database tables to application objects. To face the issues behind the impedance mismatch, some dedicated solutions were devised.
First of all, to eliminate the relational-object "gap", a new database model has emerged – that aimed to eliminate the relational part – and treated the data in an object manner. A couple of new object [10] and the object-relational databases have been developed, providing native object-oriented languages, alternatively to purely relational SQL. Due to the fact that object databases got integrated with the object-oriented programming language, the same model of data representation guaranteed the *consistency*. Object Oriented Database Management Systems (OODMBS) become especially useful for storing complex data, as there was no need for columns and rows. Instead the relations were preserved directly between data objects. Moreover, *many-to-many* relations could have been achieved using pointers that were linked to objects and thus, established this kind of relations between them.
On the other hand, in the early 1990s, the object-relational DBMS (ORDBMS) chose to take some of both, relational and object world. The differences between relational and object-relational DBMSs deserve a detailed distinction (see 2.3.1). In general – by contrast to the object databases – while using the mixed model, data ought to be stored in database and it should be manipu-

[9] Specifying Queries As Relational Expressions
[10] MUMPS (1966), Gemstone (1982), Versant (1988), db4o (2000)

lated with query language. Thus, the object-relation model would have to consider keeping the declarative query language while adding object concepts.

Despite proliferating new models, the relational model has occupied its predominant position as of the enterprise applications of databases – at least until present. A comprehensive overview of the modern data storing solutions will be presented in the following sections.

2.2.2 Revolution - *the Web changes everything*

Not Only SQL. Apart from immense growth of relational databases and their massive adoption, the first research on object and – even more surprisingly - NoSQL based database is dated to 1980 and 1966 [11] respectively. However, since the early (as already mentioned – 1966) NoSQL [12] introduction, the situation has changed dramatically and forced massive adoption of arising NoSQL models in 2000s. This was due to Internet access widespread and the fact that it has become part of everyday life for most of the civilized world. The amounts and sizes of data that had to be collected and stored grow exponentially (see Figure 2.1). The augmentation of new

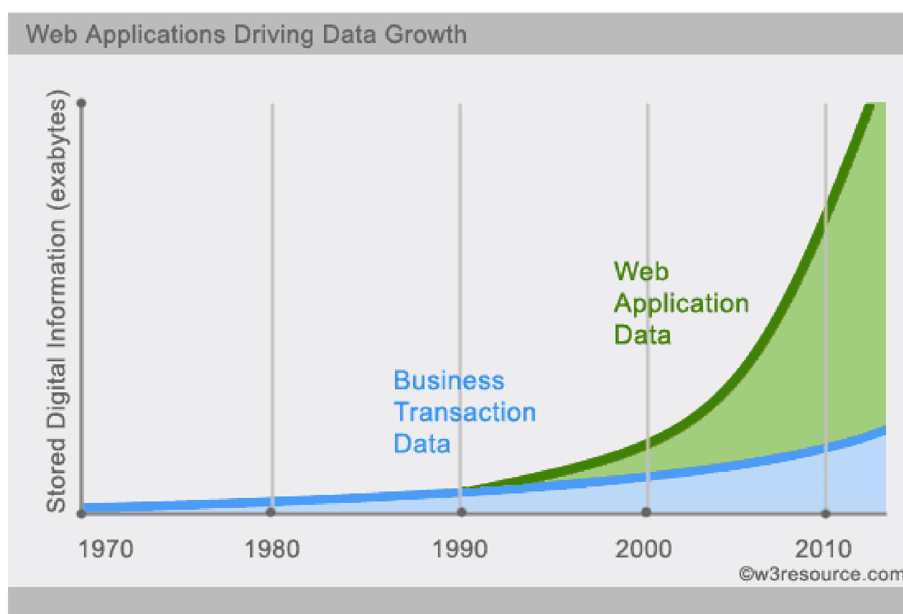


Figure 2.1: Increase of web data storage.

data and its increasingly complex nature that also often had to be changed dynamically and accessed instantly, caused rapid development of dedicated and highly specialized NoSQL data models. Responding to this demand, post-relational data stores aimed at specific needs of the systems that required rapid data delivery. This resulted in compromise solutions between the traditional database models and the new, NoSQL approach [13] . The NoSQL concept came up as a result of a few reasons due to Internet proliferation. Firstly, the present relational databases could not easily scale (i.e. system ability - as discussed in [32]) to handle unprecedented data set sizes.

Scalability The scalability term is a system property that is quite complex and thus, difficult to form in a concise definition [33].

[11] MUMPS (Massachusetts General Hospital Utility Multi-Programming System) or alternatively M, provided ACID (Atomic, Consistent, Isolated, and Durable) transaction processing with build-in, schema-less database.

[12] NoSQL was first used by Carlo Strozzi in 1998. It was used to name his Open Source, Light Weight, DataBase which did not have the SQL interface.

[13] As there is no strong definition of the NoSQL concept author will develop one in following section (see 2.3.1).

DEFINITION 2.2: *Scalability*

Scalability is the desired property of a system, network, or process to handle a growing volume of data in a capable manner or its ability to be enlarged to accommodate such growth.

In particular, we can refer to the online transaction processing system or the database management system to be scalable as it enables more transactions due to new CPU or storage added transparently and in a hot-plug manner. Some other examples of scalable systems are routing protocol, peer-to-peer (P2P) architecture or distributed nature of Domain Name System (DNS). Although relational solutions allow horizontal scaling to improve *transactions per second*, but only for the requirements originated in slower networks, hardware architectures of CPUs and smaller disk sizes that were used in the past.

INFOBOX 2: *Scaling categories based on the resource add method*

- *Scale OUT/horizontally* – based on adding more nodes to the distributed system. Connections based on efficient, broad band gigabit ethernet encourage the cluster based low-cost commodity systems that can accommodate to multiple nodes.
- *Scale UP/vertically* – upgrading single node of a cluster with the use of new hardware components and improves the performance of the applications that run on the amended machine

There are some obvious tradeoffs that those models impose. The increased number of nodes also increases the complexity of management and application handling. Also complicated programming model and the throughput or latency between nodes are unavoidable issues. In general, NoSQL solutions sacrifice some of the relational virtues to meet the challenge of the new modern computer world demands like web-scale data volumes or scalability. Due to the Web based nature of the modern data, its distribution and requirement for partition tolerance have also become serious challenges for the traditional database solutions. RDBMS has ACID and supports transactions, therefore scaling out with RDBMS is harder to implement due to these concepts. On the ground of growing demand for fast access to large amounts of data, new models that prefer loosening the tight relational rules, has emerged. NoSQL solutions have introduced specific rules, adequate for newly emerging web solutions and their data access requirements.

General expectations while considering web based and thus, distributed databases have been put in a concise form of *CAP Theorem*.

THEOREM 1: ***CAP Theorem (a.k.a. Brewer's theorem)***

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- **Consistency**: all nodes see the same data at the same time
- **Availability**: a guarantee that every request receives a response about whether it succeeded or failed
- **Partition** [a] **tolerance**: the system continues to operate despite arbitrary message loss or failure of part of the system

[a] Partition should be considered not as a discrete switch but rather as a probability function based on sub-upper bound of latency. If a request times out we call it a partition and move on and do something in response.

However, the problem with NoSQL solutions is that mostly they support only reduced level of consistency - so called *eventual consistency*. The CAP theorem is often considered as justification for this fact. Design assumptions of NoSQL pushed out the problem of strong consistency to the application business logic. In this manner NoSQL consistency is sacrificed in the name of high availability and decreased system load. On the other hand, assuring the *eventual consistency* also increases cognitive complexity of distributed applications for developers and users.

DEFINITION 2.3: ***Eventual Consistency***

Eventual Consistency – A limited version of consistency assurance that assumes that replicas of every single record across the distributed nodes of integrated grid can diverge in their value for some period of time.

It means that even if all operations stop, it will take some time for the system to make all the copies of replicated data the same. This makes it a complex problem to assure the data up-to-date state. Such assumption makes consistency a purely liveness [14] [34] guarantee because the consistency might only theoretically occur at some time after execution ends.
Despite the present NoSQL revolution, one has to be aware that still, there is a major need for stable and reliable, fixed schema databases. It is important to take notion that the NoSQL only complement the relational model in the cases when relational strict rules become obstacles for reaching the goal.

Nowadays we are witnessing an evolutionary step forward that merges the best from the relational and NoSQL worlds. But before we can discuss it, we should consider the enterprise logic that has initiated rise of this new approach. Basically there are three trends that have driven the evolution of thinking about data over the last few years.
First, fairly obvious - and already mentioned – is that data sizes are getting bigger. This is caused by collecting and keeping more data that becomes available due to Internet popularization.
Secondly, since different data sources have been brought together and because the data tends to evolve more quickly presently we are experiencing a kind of de-emphasis on modelling data with schema and being strictly formal about it. In the same way we recognize code modelling as counter productive if taken too far with the UML object diagrams.
Finally, migration to the data-driven applications, and by this eliminating the need for imple-

[14] Informal requirement toward the distributed system that some positive characteristics will eventually be achieved.

menting the domain behaviour from the application code, should be considered.

The designers of relational solutions did not foresee such intense progress of hardware technologies. Bigger SLA [15] requests were solved simply with more hardware not considering the horizontal scaling. Yet the leading enterprise, internet companies perceived the horizontal scaling of traditional relational databases to be too difficult or too expensive. In some cases, traditional, relational solutions could be sharded to meet the needs, however it usually occurred to be too expensive in terms of energy costs and a number of servers to achieve the expected performance and goal of the system. An interesting solution to distribute such data could be simply adding disks working with data on the affordable [35] and distributed file system, just as *Hadoop Distributed File System* (HDFS). However, the ideas to face such problems also considered some trade-offs, like promoting availability at cost of eventual consistency (accepting transient inconsistencies that could be fixed post-hoc if needed) and neglecting the need to store everything in the relational database. It occurs that some data can be stored with evenly efficient results as non relational models like: key-value stores, document stores (XML or JSON) or column stores. For instance the document store has proven its fitness to store deeply structured and nested data and therefore became a convenient way of storing semantic knowledge about data structure, embedded in the system due to XML or JSON. Another example considers denormalizing data and storing it in multiple columns. Due to high cost of `JOIN` operations a column store can cheaply get the columns that we are interested in, without the need to read entire rows and extracting the requested data.

In general those emerging issues are often gathered under the *Big Data* term. The Big Data is all about integrating different model data sources, taking data as they are and integrating them into one, fixed integration schema. However, it must be noted, that this class of problems applies to data stores that are more than just a few terabytes of data. In such a case the architecture of an application using Big Data should minimize the domain knowledge in the code and focus on the actual data that is already present in heterogeneous, NoSQL data sources instead of trying to model every aspect of the data.

Now let us compare and contrast the possible solutions of NoSQL and storing data in the distributed file system like HDFS (e.g. HBase providing BigTable-like capabilities for Hadoop). The DFS as a general purpose file storage gives more flexibility about how the data is being structured. It is true in terms of its internal representation on the system and in terms of structuring data in directories, and how they are going to be shared. It is most suitable for table scans and writes with contrast to rather being unfriendly towards CRUD operations with individual records. However the NoSQL – as a database persistence – is not as scalable in terms of cost per TB and speed of scanning but it is handling CRUD operations very well. Therefore the choice of the particular model must be made carefully and it should be considered which solution suits the data best.

For example: moving the transaction logic to application, using key-value store while parsing the value BLOBs into objects or writing queries in JSON instead of SQL - are all signs of serious mismatch between the data requests and used data model.

STATEMENT 1: ***Data-model matching requirement***

Data storage model must be adequate to the stored data future appliance and data request cases.

Regarding the web nature of current computer systems, the modern solutions would have to consider the *On-line Transaction Processing*. OLTP facilitates and manages transaction-oriented

[15] A service contract defining the service quality conditions and guarantees; e.g. contracted delivery time of service from service provider to service client

applications for data entry or retrieval. OLTP was used to handle client transactional request (money withdraw, order item from shopping cart, etc.) with RDBMs. Since statistical and business reasons, such transactional OLTP data have to be consolidated. For this purpose the *Extract-Transform-and-Load* (ETL) tools are used to convert a collection of OLTP systems to a common format and load it into one or more *data warehouses*. *Business Intelligence* (BI) queries towards data warehouses are rather heavy and stale due to the data being supplied to data warehouses periodically from DBMSs. These circumstances are not acceptable for the OLTP application main goals of availability and throughput (speed [16] and also concurrency and recoverability) and can not assure timely transaction responses. Such combination was working for the pre-web intense applications. The current Web-based (social networks, online gaming) and smartphone sensor applications, however increase the volume of interactions with DBMS and require far more OLTP throughput with better DBMS performance and enhanced scalability. Moreover, due to the intense data input and usage a real-time analytics becomes also crucial. This is especially substantial to e.g. electronic trading or augmented reality smartphone applications that require maximum of real-time inquiries to the current data. On the other hand, the costs of data warehouses extensions and upgrades are high. Thus often data warehouses for large internet companies store only the data for a limited period of time e.g. couple of months and then have to purge it because upgrading its capabilities would involve million dollar investments. On the whole, the data warehouse has mature, rich SQL analytic functions but only for scaling mid-sized [17] data. On the other hand, the Hadoop is a way cheaper solution as of cost per TB of data (see [37]) and scale well with Big Data petabyte sizes.
Now the traditional, SQL-based OLTP architecture capabilities can be exceeded by workload of modern OLTP needs and moreover, the outdated data stored in the data warehouse becomes useless while considering real-time interactions. On the other hand, the NoSQL solutions with their non-transactional approach does not provide the real ACID and pushes the responsibility for it to the application programmer. What is more, the analytical and non-programming experts use SQL, that NoSQL solutions in general lacks.

The solution was to move data from warehouses into Hadoop which was less mature but supported massive scalability with the orders of magnitude less costly per TB. The main issue, however, was that Hadoop, as an exemplary solution, lacked native SQL. This has changed when the *HIVE* – providing a data warehousing infrastructure with the SQL-like syntax query language (HiveQL) and analytics build on top of Hadoop – came in [18] . This was crucial as SAP, marketing or statistical experts were not programmers and needed the SQL. However, HIVE was based on *MapReduce* that has proven not to be efficient even with medium result sets. This resulted in many alternatives like SparkSQL, Impala, Presto, Drill etc. has emerged (See section 2.4.3).

2.2.2.1 NewSQL - the Evolutionary Step

Recently the data storage evolution has moved to a stage where, what would seem to be the best solution, is to acquire high performance with scalability, while sustaining the traditional, transactional ACID. The response to such problem would be the most recent – as of 2015 – database evolutionary step – *NewSQL* [19] databases. The NewSQL solutions combine the high throughput of NoSQL approach and preserve the internal consistency assurance mechanisms in the form of real ACID. What is more, NewSQL would employ SQL as the primary mechanism for application interaction and should represent scale-out, shared-nothing architecture, capable of running on a large number of nodes without bottlenecking.

[16] Query Throughput is a classical metric measure characterizing the ability of the system to support a multi-user workload in a balanced way. This is often used to determine the performance of a database system [36].

[17] I.e. less than a petabyte.

[18] Published in 2009 in white paper [38] by the Facebook Data Infrastructure Team and elaborated in [39, 40].

[19] The term was first coined by Matthew Aslett in [41] report.

⋆ ⋆ ⋆

This short review of existing data storing solutions and the diversity of their functioning mechanisms has established a need for blazingly fast unified access, integrating middleware. The requirement also states fast access regardless of dynamically changing data and structure. The requested solution would also have to be extremely elastic and easy to modify. On the other hand, such middleware needs to face potential distribution issues across integrating data sources. Even more problems arise when we consider middleware that works with distributed and heterogeneous – i.e. different origin – data sources. The distribution and data source particularities have to be taken into careful consideration in the middleware development process. In the following sections integration rules and crucial issues, that must be solved, will be elaborated.

2.2.3 Integration - Principia and Taxonomy

Challenging new data processing issues of large volumes of data or real-time stream processing of unstructured data have resulted in proliferation of the purpose-dedicated data storing solutions. A long time perspective of using different and numerous data sources, and the need of absorbing its data for central querying and analysis have forced some ideas (like *federated databases management systems* (FDBMS) or *multi-database* [20]) for data integration.

2.2.3.1 Data Integration - DI

The ultimate goal of *data integration* is combining data – stored with the use of various technologies – from two or more disparate data storing sources into one, end-user unified data access interface. This process involves commercial, industrial and scientific domains. The need to share the existing data from the proliferating data sources and their great volumes (See section 2.3.7 for numbers) made data integration a very important aspect of modern data processing and analysing computer systems. The data integration is a growing market and one of the major challenges for the future of IT. This is mainly due to two contexts: the internal organization data integration and the inter-parties data integration. Moreover, it must be noted that under *data integration* term one can refer to several sub-areas of interest – such as:

- *Data Warehousing* (DW) – As a central repository of data from disparate integrated sources, used for reporting and data analysis. (See also section 2.4.1.1
- *Data Migration* – Simply a process of transfering data between two or more storage systems. In terms of the *Extract-Transform-Load* (ETL) processing at least *extract* and *load* phases must take place. Most often present while migrating between hardware architectures, databases or data-based applications.
- *Business Intelligence* (BI) – The process of interpreting the data context. Making raw data an information rich analytical resource for supporting decision making. Sometimes also referred to as *Decision Support Systems* (DSS). BI involves data visualization, data mining, reporting, time-series analysis with predictive techniques (behavioural prediction in time), On-line Analytical Processing (OLAP) and statistical analysis. (See also section 2.3.7)
- and *Master Data Management* (MDM) – A paradigm that proves its value especially in large and diversified environments like big corporations, based on linking/storing all crucial system data [21] in one file – master file. The master file serves as a point of reference. The central file can facilitate computing in various system applications, platforms and

[20] Comparing to FDBMS less integrated, but owing to middleware supporting distributed transactions across the participating databases used by a single application

[21] E.g. business objects for transactions. MDM is also complementary to BI and can provide an excellent source of dimensional data for analysis and the analytical data itself – thus supporting decision making.

architectures. It can be stored in central hub, data warehouse or every application can have its own MDM that is later merged across applications with a central registry

In general, the *data integration* techniques can be applied in different forms. Let us point them from the most manual to the most automated.

- *Manual Integration* – the user has to deal with integration involving accessing all data sources using *Common User Interface*. However, the data here has no unified view.
- *Application-based Integration* [22] (AI) – The integration is handled by application. This level of integration is sometimes referred to as *Enterprise Application Integration* (EAI) (e.g. TIBCO). This type of integration is used as a link between multiple applications at the functional level. It is focused around transactional or service call level. This is transaction-aware software, thus it can proceed with one work unit (e.g. employee creation) decomposed into many diverse pieces in various data storing sources. Such integration deals with integrating live operational data in real-time between two or more applications. The AI is event-based, due to providing as-soon-as-possible rise of a transaction across integrated applications when a new request is commenced. This assures real-time reactions and most current system synchronization. The main effort has been put on real-time performance, transactionality, reliability, and once-and-only-once guaranteed delivery. This is due to the fact that AI applications have mostly come into existence from the message-queuing and the *Enterprise Service Bus* (ESB) (see section 2.4.4) products. AI is best suited for operational transactions while the DI is made for analytics. The most modern incarnation of *Application Integration* is the *Cloud*-based approach (e.g. MuleSoft), as a result of general trend towards cloud. The biggest advantages of AI are its agility to accept fast changing requirements of integration:
  - Instant data – No DI batch processing. It involves moving toward time effectiveness of ESB level – i.e. seconds.
  - Data *Software as a Service* (SaaS) – Moving from explicit database interactions into the cloud-based SaaS application API
  - Agility – The need to provide elastic approach to data and logic changes in favour of application based data access due to ease of modification of well designed application
  - Cloud Application API – no need for complicated data center support in the case of Cloud-oriented storage applications
- *Middleware Data Integration* – Moving the integration logic from applications to one integration layer
- Virtual Integration – The data is kept in the source systems but an integration views are developed providing unified and transparent access for the end-user. This approach brings near to zero latency for the data updates on the integrated data sources to propagate across the integrated grid. Moreover, there is no need for the additional storing infrastructure. The disadvantage is difficulty of version managing after data updates [23].
- Physical Data Integration – Creating a copy of all integrated data in the central repository (e.g. data warehouse). Fast access, analytics and ease of data manipulations are the main benefits, while on the other hand, a large data center must be assured for the large amounts of data to be centralized and handled.

The most general case of data integration involves one, unified user query towards the DI global/integration schema, and a set of legacy/integrated data sources' mappings that are

[22] In contrast, DI is batch and unaware of concept of transaction. The DI software, most often is an evolutionary step that has originated in the ETL tools. The DI also deals with standardization, validation, transformation, synchronization and mapping of large data volumes. Owing to physical data abstraction enables the data access.

[23] Solution proposed in this dissertation overcome this drawback.

handled and transformed into real data, required by the user query. Techniques developed to meet the integration needs involve:

- *distributed database systems* – utilizing homogeneous data sources managed by distributed DBMS
- *tools for source wrapping* – including the *mediator* based architecture with global schema and mapped heterogeneous and autonomous data sources
- *federated databases* – utilizing heterogeneous and autonomous data sources (i.e. DB2 Information Integrator)
- *distributed query optimizations* – used in *P2P* network of autonomous data sources mapped with each other, without global schema

Moreover, a well designed DI solution must comply with some basic characteristics:

- Location & Access Transparency – no physical notion and characteristic of the actual source data are supposed to be revealed to the end-user
- Heterogeneity – variety of multiple diverse data sources should be supported (see also section 2.2.6.1)
- Extensibility – there always should be an easy way to plug-in a new data source without need of architectural changes
- Performance – solutions should be ready for petabyte-size data volumes to be integrated as a sum of all data sources
- Autonomy – integrated data sources should work the same as before becoming part of integrated grid, without any disturbance

More complex than the structural transparency of location or access methods for integrated data, is to assure the logical transparency of integration schema. The integration schema (or ontology) should provide conceptual view, independent of contributory data schemas of the data sources. Thus integration schema is a formal meta-description of the schema that the end-user actually queries. Transformation of the data source contributory schema views into the global, integration view is done with a set of mappings that also requires some formal specification.

Such requirements are most often implemented with one of the approaches: mediated architecture, global view materialisation-based data exchange and P2P data integration. The data exchange however, in contrast to DI, considers data restructuring with possible data loss of content (i.e. no/many way(s) of transforming an instance of a data with given constraints) and central data store due to global schema materialization. On the other hand, the P2P solutions also bring some disadvantages, while each peer using local and external sources with distributed queries, which in turn causes the network overhead in query-intense environment, especially with high volume data transfers. Due to the mentioned drawbacks of the two approaches, this dissertation is focused around the mediator-based architecture.

Enterprise Information Integration (EII) In general the *Information Integration* (a.k.a. referential integrity - see 2.1) merges information with diverse conceptual, contextual and typographical representations. The goal is to combine information from the data stored in different sources, and thus to reduce information uncertainty.

Regarding the commercial data integration the *Enterprise Information Integration* (EII) has been a goal for multiple - mainly big - companies. Its target is to provide unified data access as a data abstraction over multiple company's resources, stored in multiple data sources (like: a large number of RDBMS varieties, text files, XML files, NoSQL stores, NewSQL stores, spreadsheets, etc.) all involving various, dedicated storage, indexing and data access methods, that are also often proprietary. The uniform data access requirement, on the other hand, assures the unified connectivity and data control across the data sources. Such a unified information

representation, regardless of its discipline or realm, covers the domain impedance and enables the data to be displayed and processed, as if there was no difference in its origin. Different methodologies or metrics for data collections stored in the target sources are represented as a single collection conforming integration schema filled with raw data without legacy sources heritage characteristics.
As an enterprise solution the integration method must confirm its maturity. There are some characteristics that can help to conform that a software is EII-ready.

- *Loose Coupling* – System must have its components designed with little of no knowledge of the definitions of remaining components of the system. Lack of direct knowledge between the components makes the system more elastic.
- *SOLID*-based [24] – Implement basic principles for the object oriented software (if implemented using the object-oriented language)
- Disparate data sources ready – The integrated data sets should be complete and accurate according to the configuration based on commonalities between data sources. Each data source may have a commonality that can be used to implement disparate data joining rule (e.g. primary key, object ID, virtual best row ID etc.)
- Lightweight – as integration usually deals with massive amounts of data the overhead provided by the middleware integration software should be minimized

Some of the EII most suited technologies are *ODBC, JDBC, XQJ, OLE DB, ADO.NET, XML-XPath-XQuery,* etc. The integration goal has been approached in multiple ways. Let us then discuss some of the most significant ones.

2.2.4 Data Integration Practices

The service that could provide unified and transparent access to the collection of data stored in multiple, autonomous and heterogeneous sources – in the form of data integration solution have been an ultimate goal for many years now. There have been three architectural approaches for data integration based on the data warehousing, mediation and federating.

Data Warehousing Based on the approach that data sources are translated from their local schema to a global schema and copied to a central DB. The first research for this kind of data integration started in 1981 [42] and since then the integration was based on the *Data Warehouse* ETL-based solutions. A single data repository made query resolving fast, but on the other hand – brought the problem of *tightly coupled* [25] (see also section 2.2.5) data architecture. Such an approach becomes a real problem, especially when considering data sources with frequently updated data, that requires the ETL jobs to perform a continuous synchronization with datasets [43].

Mediation Some evolutionary changes have been considered with the use of mediated schema and real time access to the source data [26] . This is also often referred as *virtualization, schema translation* or *mediation, lazy integration, database federatio*n, *Information-as-a-Service* (IaaS) or – as already mentioned – *Enterprise Information Integration*(EII). The general idea behind this approach is to integrate heterogeneous data sources without the need of ETL jobs. Owing to a common, global *integration schema* definition, the end-user will have a unified view to the integrated data. Each integrated data source requires a wrapper that enables its data access and an execution engine/architecture. The user query against the integrated schema is then

[24] Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion
[25] Considering more: interdependencies, coordination and information flow.
[26] This is the approach that will be elaborated in chapter 3 as the main idea for the dissertation topic integration architecture developed in 2009 by the author.

decomposed into multiple sub-queries understandable for each data source participating in the integration. Then each of the sub-queries is sent to the wrappers that by-pass the queries towards the data source itself. The result sets are then transformed into the *integration view* according to the designed strategy. Thus all the time requested data is up-to-date and does not require one central data repository to be filled in contrast to the earlier DW and ETL based integration. The lack of one central integrated data store, on the other hand, resulted in long response times due to the network and data transfer overhead. According to [18] there are two main approaches of global schema to the source schema mapping (*Global As View*) and the source schema to the global mapping (*Local As View*) – see section 2.2.5 for details.

Federation The mediator approach forces the use of some central unified schema that is a central point of failure. However, solutions based on the federated architectural model have overcome this problem by providing a component based model of centralized or distributed (Peer-to-Peer based) nature. The goal of federated architecture is to provide:

> “(...) mechanisms for sharing data, for sharing transactions (via message types) for combining information from several components, and for coordinating activities among autonomous components (via negotiation).”
>
> – *Heimbigner and McLeod (See [44])*

Moreover, component interactions in the federation are managed due to the export schema and the import schema. The information that each component will share with other components is specified with the export schema, while the import schema specifies the non-local information that a component wishes to manipulate[44]. Therefore the federated approach provides all its components (i.e. autonomous and heterogeneous databases) a notion of data materialisation. This way when a client queries the federation, the system knows which component stores the requested data and passes the request to it. In the case of heterogeneous components the federation already must be, prior to the query execution, configured to handle multiple queries and automatically combine the results.

However, it should be noted that the federated databases have a couple of drawbacks, regardless if their nature is centralized or distributed. As data is scattered across all data storing components of the federated architecture single failure of a component can cause an issue just in the same manner as a single point of failure in centralized architectures. Moreover, as a distributed architecture a single component latency will cause the entire system call delay. This results in the additional effort to program the applications to consider incomplete query results in the case of such time-outs of component database. The case when all sources in federation must communicate also results in network overhead of n^2 mappings. The issue is getting even more complicated if the sources are dynamic, and therefore need constant mapping changes. Additionally, this architecture requires an initial configuration at start, and maintenance during functioning of such a federation, thus increasing overall costs.

⋆ ⋆ ⋆

However, the architectural approaches for structural integration has occurred insufficient in cases when integration involved related domains that describe the same data with different terms. Overcoming such issues has forced devising *ontologies*.

Ontologies Due to constant evolution and more advanced integration implementation requirements a new semantic-based problem has arisen. As the data structure – while being integrated – still, might vary not only with a structure, but also with the semantic context of its content. In other words, the architectural structure might be consistent, but the semantic

interpretation of its content might be different due to collisions of semantic understanding. For example, the concept of *profit* can be considered as a monetary gain or in another case, as a number of transactions. This set of semantic collisions has been considered as a target for *ontology* – i.e. *"formal and explicit specification of a conceptualization"* (see [45]) – based solution. This includes:

> “ The objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold among them. ”
>
> – *Encyclopedia of Database System (See [45])*

Ontologies allow to solve a semantic issue due to their role of describing (formally specifying) the concepts and relationships for a given domain, and thus also the architectural components of integrated data view. However, ontology is not only limited to traditional perception of definition that introduce terminology of a domain without any knowledge. Ontology also states axioms [27] to constrain interpretation of a defined term [46, 47]. At the heterogeneous information level, ontologies provide unambiguous entity identification and assertions for named relationships that connect those entities. Thus, explicit definition of terms and relationships in ontology supply accurate data interpretation in context of numerous data sources. Moreover, ontologies can play a role of global query schema (where queries are being build), and finally mappings between various data sources schemas can get verified with the use of ontologies [48]. The meaning of ontology in data integration is focused around explicit description of data.

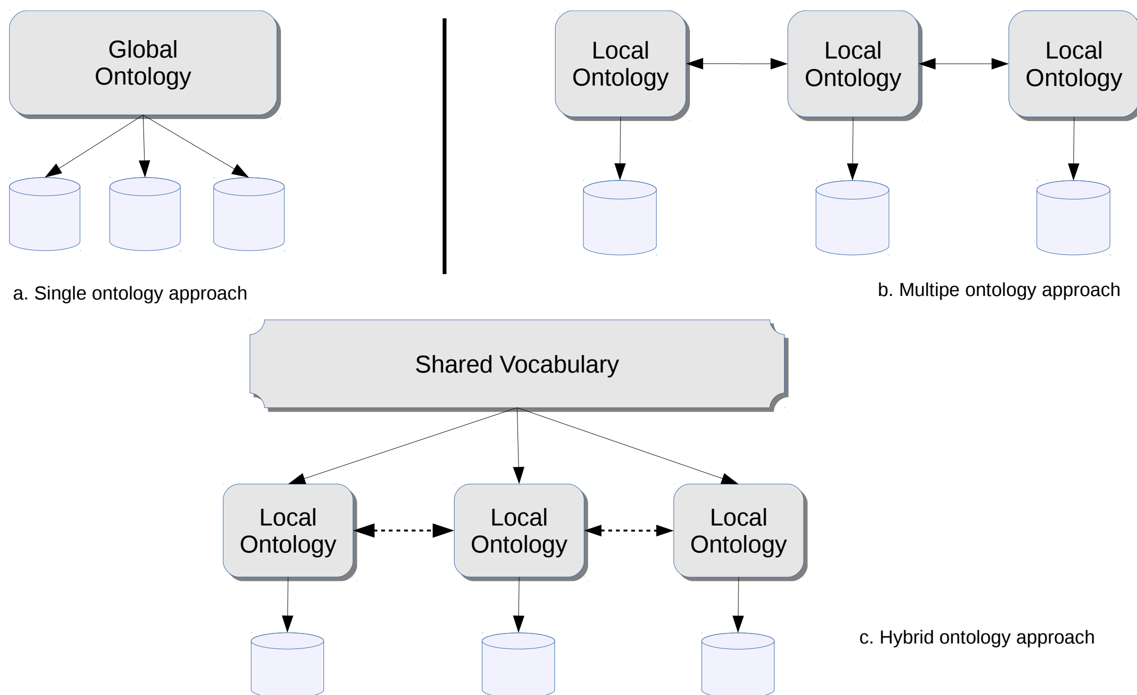


Figure 2.2: Ontology types.

Due to semantic complexity of the data, ontology-based integration evolved into three main ontology approaches: *single ontology, multiple ontology* and *hybrid* 2.2. Now the *single ontology approach* uses one global ontology that shares the semantic terms *vocabulary* [28] . This means

[27] Assertions with a logical form of rules. Axioms include only statements asserted as a priori knowledge. Moreover, axioms cover the theory derived from axiomatic statements.

[28] *Vocabularies* are used to describe and represent an area of concern by defining the concepts and relationships (also referred to as "terms"). They are used for classification of terms used in particular application, characterise possible relationships or define possible constraints on using the defined terms. No clear discrimination between

all of the information from the integrated data sources are described with one global ontology. Such feature is especially important for modularization purposes. This means that each *domain ontology* describing concepts of a specific domain in a uniform way, can also enable generalization when two distinct ontologies merge, based on some foundation ontology. Which is especially important while designing one, large and monolithic ontology. Single ontology should be applied for the cases dealing with integrated information that considers the same domain aspect. The drawback is that it lacks flexibility for data source changes that can affect the domain concept.
An answer to this issue are the *multiple ontologies*, where each ontology models an individual data source, and thus, while combined can be used for integration. In this case, each data source has its own ontology that does not share common domain vocabulary among other data source ontologies. However, a single data source ontology can be composed of many local ontologies sharing the same, local domain vocabulary. This approach does not require a foundation, minimal and common ontology for all data sources. Each ontology is autonomous and related only to local data source, and thus immune to local domain changes. However, a lack of common domain vocabulary shared across multiple data sources ontologies makes the inter-ontology communication difficult. Thus, the additional formalization in the form of mapping that expresses similarity or identical nature of separate data sources ontologies is required [49].
Finally, the *hybrid ontology* attempts to face all of the issues from the two previous approaches. The hybrid approach is the same as the multiple ontology approach, however, it utilizes shared vocabulary across all integrated data source local ontologies. The local ontologies must use the shared vocabulary to be build. The shared vocabulary cover basic domain terminology. The vocabulary itself can sometimes be described as an ontology. The great advantage of this solution is simplicity of adding new data sources without the need to interfere with the global system. Moreover, evolution of ontology is easy to perform and reduces the need for existence of the mappings between local ontologies. The most important disadvantage of this approach is that each data source ontology cannot be reused and must be rewritten from ground up.

Ontologies must be formed with a representation language, as an intended semantic level specification, and thus independent of data modelling strategy or implementation. One of the most popular languages is *Web Ontology Language* (OWL) as an evolution of RDF knowledge representation data model (see 2.3.5.2). It enables making ontological statements for *WWW*, its classes, properties and individuals defined as the RDF resources with the use of *RDF Schema* and identified with *URIs*.

Across multiple ontology languages most of them encodes common components of ontology such as: Individuals, Classes, Attributes, Relations, Restrictions, Events, etc. See Listing B.1 for exemplary OWL/XML syntax for an ontology.

Listing 2.1: OWL/XML Syntax for Ontology Management

```
1  <!DOCTYPE Ontology [
2      <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
3   ]>
4   <Ontology
5     xml:base="http://example.com/owl/families/"
6     ontologyIRI="http://example.com/owl/families"
7     xmlns="http://www.w3.org/2002/07/owl#">
8     <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
9
10   ...
11  </Ontology>
```

ontology and *vocabulary* has been settled. One is mostly accustomed to using the term – ontology – for more formal and complex collections of terms, whereas the *vocabulary* is used when loose or non formalization is required. E.g. in the case the book author can be related with two distinct relations (say "author" and "writer") in integrated grid, a very simple vocabulary would be used here to express the identical nature of both relations

This ontology-based data integration allows to integrate information about data that can be also classified as the *GaV* approach. This is due to possibilities of unambiguous identification of entities from the integrated sources and assertion of applicable named relationships used to join such entities. Effectiveness of such integration, however, depends greatly on how expressive the ontologies are and how concise their domain perspective is.

2.2.5 Integration Theory

To formalize the data integration theory – as a subset of database theory – the *first-order logic* (a.k.a. *first-order predicate calculus*) formal system can be applied. To describe the difficulty and complexity of data integration some abstract and general definitions can be settled.

Considering a theoretical approach for data integration one can formalize an integration system according to Definition 2.4.

DEFINITION 2.4: *Data Integration System* - DIS

A data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where

- $\mathcal{G}$ is the integration (a.k.a. mediated or global) schema.
  $\mathcal{G}$ is expressed with $\mathcal{L}_\mathcal{G}$ language over $\mathcal{A}_\mathcal{G}$ alphabet. $\mathcal{A}_\mathcal{G}$ contains symbols for every element of the $\mathcal{G}$ schema [a]
- $\mathcal{S}$ is the heterogeneous source schema.
  The $\mathcal{S}$ schema is expressed with $\mathcal{L}_\mathcal{S}$ language over $\mathcal{A}_\mathcal{S}$ alphabet. $\mathcal{A}_\mathcal{S}$ contains symbols for every element of the source.
- $\mathcal{M}$ is the mapping between $\mathcal{S}$ and $\mathcal{G}$
  $\mathcal{M}$ is created based on the following transformations:

$$\begin{gathered} q_\mathcal{S} \rightsquigarrow q_\mathcal{G}, \\ q_\mathcal{G} \rightsquigarrow q_\mathcal{S} \end{gathered} \tag{2.1}$$

  where, $q_\mathcal{S}$ and $q_\mathcal{G}$ are two queries of the same function over the source $\mathcal{S}$ and global schema $\mathcal{G}$ that *"leads-to"* each other.

[a] e.g relation if $\mathcal{G}$ is relational, class if $\mathcal{G}$ is object-oriented, etc.

Now the $\mathcal{M}$ is a set of assertions that are used for semantic translation between $\mathcal{S}$ and $\mathcal{G}$. The *assertion* here, is a statement $Concept_X \rightsquigarrow Concept_Y$ that instructs that the concept expressed on schema X is the same as the concept on schema Y.

2.2.5.1 Schema Mappings

Regardless of architectural approach for data integration there is always a need to represent data in a kind of unified form. For this reason some global unification patterns can be applied.
As already mentioned there are a couple of types of $\mathcal{M}$ mappings. *GAV*, *LAV*, their hybrid – *GLAV* and decentralized *Peer-to-Peer* approach. Let us discuss some of the main features of those mapping approaches.

Global-As-View mappings (GaV) *GaV* can be described as a limited view over the data. Here the mediator schema, acts as a view over the source schemas. This involves rules that map a mediator query to source queries. Likewise regular views, the accessible view perspective through the mediator is a subset of actual available data from sources. This means a set of

queries on local resources $\mathcal{S}$ (with real data), one for each element $g \in \mathcal{G}$ – of global schema. In other words, GaV represents mapping model with the assertion of mapping elements, that associates a query over source schemas to each element of mediated (integration) schema. Thus, mapping defines exactly how the element g is computed from the local source. In formal notation GaV connects in mapping $\mathcal{M}$ each element from the global schema $\mathcal{G}$ with the query characteristic of schema $\mathcal{S}$. Thus, query language $\mathcal{L}_{\mathcal{M},\mathcal{G}}$ allows expressions that are build based on $\mathcal{A}_{\mathcal{G}}$ alphabet. Thus the GaV mapping ($\mathcal{M}$) is a set of assertions , one for every $g \in \mathcal{G}$

$$\forall_{g \in \mathcal{G}} g \rightsquigarrow q_{\mathcal{S}} \tag{2.2}$$

The GaV model is based on defining global schema as a set of views based on local schemas. Thus global (virtual) schema building elements contain a view for a part of adequate local schema. The mapping itself defines how to request for the local data while querying the global schema. This approach, due to its *a'priori* local access is especially efficient while considering well known and invariant, local (set-up and schema) data sources. Therefore any local data source changes or new data sources require intense effort for adopting them to the global schema.

Local-As-View mappings (LaV) In short, this mapping is a set of queries on the global (virtual) schema, one for each local source (with real data). The LaV views define how sources contribute to the global (virtual) schema. The *LaV* mapping approach connects every element of the $\mathcal{S}$ schema, with a query characteristic of schema $\mathcal{G}$. In other words, the query language $\mathcal{L}_{\mathcal{M},\mathcal{S}}$ accepts expressions created based on the symbols of the $\mathcal{A}_{\mathcal{S}}$ alphabet. Thus the LaV mapping is a set of assertions (translation), one for every element $s \in \mathcal{S}$:

$$\forall_{s \in \mathcal{S}} s \rightsquigarrow q_{\mathcal{G}} \tag{2.3}$$

Therefore LaV mapping expresses a type of mapping model that has assertion of mappings for every element that associates each element of the source schema to a query over the global (integration) schema.

Lav is based on the assumption that each data source content schema (local), should be characterised in terms of global schema $\mathcal{G}$ – hence local is a view.

This makes LaV well suited when dealing with already stable, well established, mature and invariant global schema – e.g. enterprise or ontology.

This approach can easily handle the addition of new data sources, that require only enhancing the existing assertions or new mapping with new assertions. All of the potential changes made to already integrated data source, can also be handled with transformation modifications.

Query processing in GaV and LaV In GaV the views of relations in the union of local schemas express the relations in the global schema, therefore this usually results in a lack of *integrity constraints* using this kind of mapping. Therefore, the mappings express the ***exact** views* – considering *Closed-World Assumption* (CWA) [29] – in the global schema.

DEFINITION 2.5: *Closed-World Assumption* (CWA)

CWA is a presumption that a *true* statement is also known to be *true*. Thus if something is not known to be *true* is *false*.

The CWA is used mainly in two situations: when the knowledge base is complete (in GaV, a set of data sources should be well known and stable), or the knowledge base is not complete and the

[29] SQL is an example of a CWA. If the query result is empty it means there are no records that fit query's conditions.

"best" final answer must be derived from incomplete information.
Owing to this *exact* representation, it is possible to process (with GaV) queries dependent on straight forward *unfolding*.

The price for LaV appliance is high overhead of query transformations between the global and local schemas. This is due to local schemas being designed based on the global schema. Therefore, views must be reversed prior to the query execution. Query processing, in the case of LaV, is based on partial views from the local schemas, and thus uses incomplete information in the global schema. Therefore answering global queries is much more complex in LaV – in terms of data and expressions – than in GaV.

Let us provide a general example of both approaches and their query processing. Please refer to Figure 2.3 for the assumed global and local schemas. The GaV based view would look like in

Local: S1_emp (Name, Age)

S1_emp	Name	Age
	John	65
	Alex	24

Local: S2_emp (Name, Age)

S2_emp	Name	Age
	John	65
	Jack	56
	Michael	33

Global: G_emp (Name, Age)

G_emp	Name	Age
	John	65
	Alex	24
	Jack	56
	Michael	33

Figure 2.3: Exemplary local and global (integration schema).

the Listing 2.2:

Listing 2.2: GaV on data sources

```
1 CREATE VIEW G_emp AS (
2   SELECT S1_emp.Name as Name, S1.Age as Age
3   FROM S1_emp
4  UNION
5   SELECT S2_emp.Name as Name, S2.Age as Age
6   FROM S2_emp
7 );
```

Now if one wants to query such integrated schema, for example for *employees* older than 50. The query would look as in Listing 2.3. However, the *G_emp* will have to be substituted – *unfolded* – (see Listing 2.4) with the definition of the view.

Listing 2.3: GaV based query.

```
1 SELECT G_emp.Name
2 FROM G_emp
3 WHERE Age > 50;
```

Listing 2.4: GaV query unfolding

```
1  SELECT Name
2  FROM
3   SELECT S1_emp.Name as
        Name,
4            S1.Age as Age
5   FROM S1_emp
6  UNION
7    SELECT S2_emp.Name as
         Name,
8             S2.Age as Age
9    FROM S2_emp
10 WHERE Age > 50;
```

The 2.4 query is later executed by the mediator because in the GaV query execution is a simple substitution of the reference to *G_emp* with the mapping in terms of local schemas.

In the case of LaV mapping, it describes the contribution of the local data sources to the expected extension of the global (integration) schema.

Listing 2.5: LaV S_1*_emp(Name, Age)*

```
1 CREATE VIEW S1_emp(Name, Age)
2 AS (
3   SELECT
4         G_emp.Name AS
              S1_emp.Name,
5         G_emp.Age  AS S1_emp.Age
6   FROM G_emp;
7 );
```

Listing 2.6: LaV S_2*_emp(Name, Age)*

```
1 CREATE VIEW S2_emp(Name, Age)
2 AS (
3   SELECT
4         G_emp.Name AS
              S2_emp.Name,
5         G_emp.Age  AS S2_emp.Age
6   FROM G_emp;
7 );
```

Let us consider the LaV mapping query on the global schema. Due to opposite to the GaV direction, no query unfolding can be done by mediator. Thus, mediator is required to reason. For example the mediator can adopt strategy that assumes fusion of the results simply by looking-up of employees names in both views.

LaV could be suitable for three classes of problems. Firstly, one could consider using LaV for the systems that integrate *data warehouses*. Instead of replicating and copying the data, it is better to create a virtual schema that integrates such large stores. Another example of LaV appliance, would be applications that are aimed to be *proof-of-concept* ones. As the LaV is a faster approach to implement – than the GaV – it fits better for prototype development. However, the best suited for LaV are the applications that require strict rules regarding the data being up-to-date.

In general the GaV approach is *procedural* as the mapping says explicitly how to retrieve data from local sources to prepare global query responses. Due to simple unfolding of the query, global query answers are computed easily. This is due to mediator following the existing rules and templates to translate the global query to source-specific, local queries. However, system extension is not a straightforward task and requires the global views to be redefined.
In contrast, the LaV approach is *declarative* as the local sources' content is described with the views over the global schema. Every source provides expressions on how it can generate pieces of the global schema. Now the mediator combines such expressions and finds all possible ways to answer global schema query. LaV can be simply extended with new local sources due to new views over global schema.

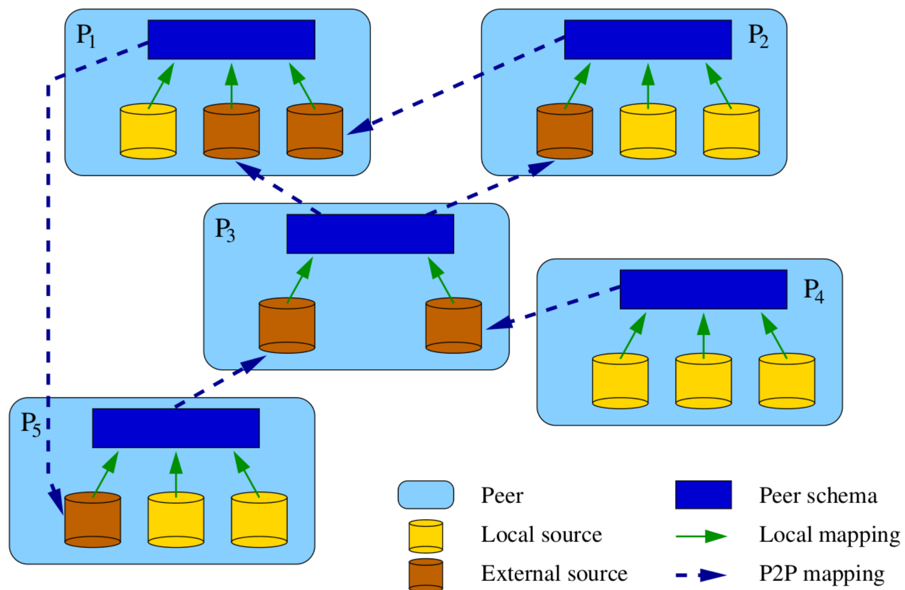
Global-and-Local-As-View mappings (GLaV) Facing all the drawbacks from the GaV and LaV approaches, a hybrid GLaV approach has been devised. The GLaV is a generalization of GaV and

LaV. In the formal form it adopts the following shape:

$$q_{\mathcal{S}} \leadsto q_{\mathcal{G}} \tag{2.4}$$

Where, $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$ relate to the queries over the local and global schemas, adequately. GLaV in a simple way can express either GaV or Lav mappings. It can be achieved with assigning queries resulting in single global relations to $q_{\mathcal{S}}$ and respectively, assigning queries resulting in single local relations to $q_{\mathcal{G}}$. Because $q_{\mathcal{G}}$ can be a query from the global point of view, the GLaV enables independent addition of new, local data sources, just as it happens in the case of LaV.

Peer-to-Peer (P2P) The distributed P2P architecture consists of nodes that are obliged to store their local schema, schema of mappings between schemas present in the P2P environment and some additional constraints. Data integration in the P2P architecture assumes that the answer to a query sent to any of the P2P participants will contain all of the requested data collected all around the P2P network nodes. The data sources are not connected to each other from the



Operations: – Answer(Q, P_i) – Materialize(P_i)

Figure 2.4: Peer-to-peer data integration

perspective of mapping between schemas. A sent query Q must be decomposed between all P2P network participants, and the partial results should be matched and sent back as in the form of full response to the end user.

The advantages of the P2P approach are decentralisation, dynamic nature and scalability. However, it also brings new drawbacks such as a lack of control over the result processing, possible long routes between the network nodes, and the network latency burden as well as connections overhead of $n(n-1)/2$ scale, with every-to-all connections.

Coupling and Cohesion As already mentioned, in the case of EII, coupling as a degree of independence between the system modules is very important not only from the point of view of an enterprise architecture but also a general assumption for discovering integration patterns. A well designed integration system should be *loosely coupled*, where each of the functional components does not need or require very little information about definitions of other components of the system. In such a case one can state that a system is well structured. The presence of high

coupling brings all the pathological issues into consideration. This involves, redundant message transmissions assuring system-wide, coupling-resulted consistency constraints, message translations, and interpretations, as well as additional validity checks. An example of a coupling-free system is CORBA that allows for object communication without the need of prior knowledge about the object implementation.

Cohesion is a coupling-related, data integration system metric. While coupling refers to the inter-module dependencies, the cohesion describes how strongly related functions are within a single software component or module. The less cohesion within a component, the more unrelated functionalities it implements, and thus grows in size instead of solving one, prior goal. Hence, the best approach is to assure functional cohesion when all module components are gathered into one module due to solving a single, well-defined task.

2.2.6 Data Integration Issues

While dealing with integration there are a couple of issues that sooner or later will become an obstacle for the system. Designing and implementing an enterprise class integration system focus should be made on real life examples, while still considering the worst case scenario. Thus an approach should consider heterogeneous data sources that are fragmented into an arbitrary pattern. Moreover, the target data must be able to be combined from numerous replications that have the same semantic meaning but are represented in different model, location, access method and present different persistence characteristics. Data integration must also consider important design consideration of where to store central-integrated data or how to organize the virtual integration approach that does not require additional resources at integrators site.

The most profound issues that must be faced while approaching the integration of distributed and heterogeneous resources are studied in the following sections.

2.2.6.1 Heterogeneity - from Structural to Semantic Mismatch

While considering central processing of such naturally diversified data a term of *heterogeneous data* has been introduced. The term *heterogenes* (*gr.* ἑτερογενής) comes from Ancient Greek as a conjunction of *heteros* (*gr.* ἕτερος, “other, another, different”), followed by genos (*gr.* γένος , “kind”); –ous as an adjectival suffix. According to *Webster’s Dictionary* heterogeneous means:

> “ Differing in kind; having unlike qualities; possessed of different characteristics; dissimilar; – opposed to homogeneous, and said of two or more connected objects, or of a conglomerate mass, considered in respect to the parts of which it is made up. ”
>
> – *Webster’s Dictionary (See [50])*

To permanently qualify the term of data *heterogeneity* in an inclusive aspect considered within the scope of this dissertation, the author has come up with definition 2.6.

DEFINITION 2.6: *Heterogeneous data*

While considering a piece of data as a part of a bigger (possibly virtual) data collection (or a superset), one should recall it as *heterogeneous*, meaning that the particular data chunk has a non-uniform technical, structural and/or semantic characteristic within considered data superset.

One should consider some basic issues while integrating heterogeneous data sources. Those problematic heterogeneity characteristics, as far as data sourcesare considered, can be categorized into three main constituents:

- **Syntactic** – Involves technical, data perspective differences. Multiple data sources in a natural way impose multiple and various *access methods* resulting from diverse communication protocols, file formats or query languages.
- **Schematic** – Additional data model diversity is provided due to various ways of *storing* the same data. Even with the same semantics, data may vary in its labels/column names across multiple data sources.
- **Semantic** (study of meaning) – Mostly occurs while dealing with the same domain data but different data schema provider. This is when we deal with related data, but different in terms of data values interpretation.
- **System** – Multiple hardware platforms and operating systems cause this type of heterogeneity

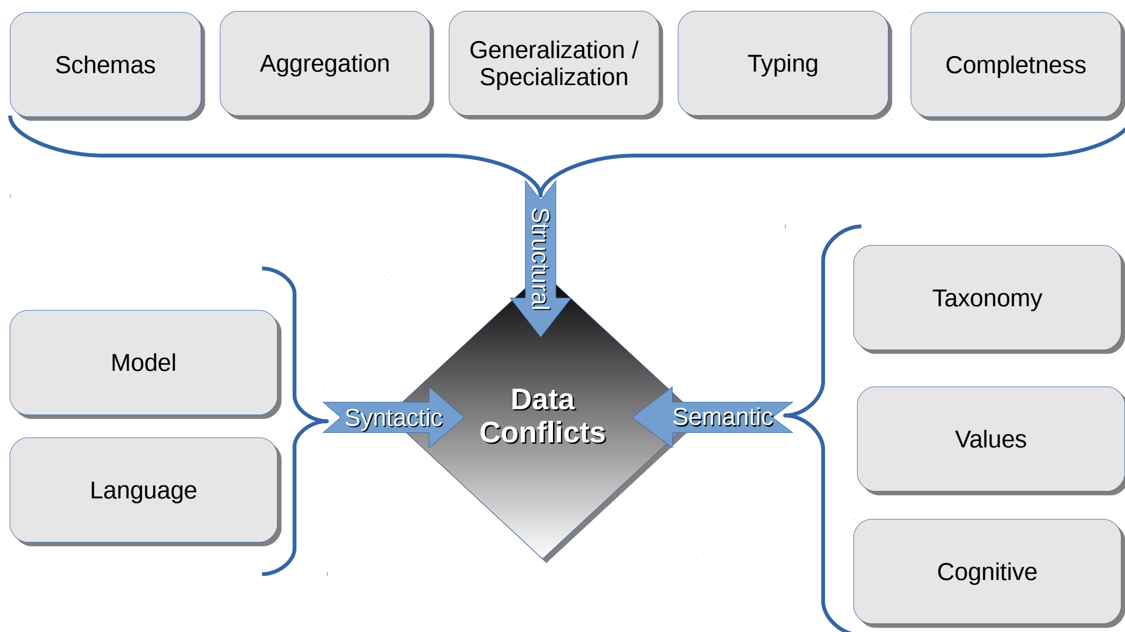


Figure 2.5: Taxonomy of heterogeneity

The most fundamental one is the semantic heterogeneity, as the source of most differences. This is due to the most often use case and challenge to integrate data systems storing data for the same or overlapping domain concepts, however, developed for vastly different business needs – thus modelled in different ways. The semantic heterogeneity requires a careful consideration to be made on mapping and transformation schemes for data uniformity. This type of heterogeneity problem has been considered for a long time now, since 1989 when there was the first data dictionary-based classification for data semantics of structural mapping [51]. In 2000 an introductory classification for *schematic* and *semantic* heterogeneity classes has been published in [52],

- *Domain heterogeneity* – raised due to semantic discrepancies in the integrated data sources. This includes schematic discrepancy, precision, unit, data representation. Solved with: contrasting the domain knowledge with the data within schema
- *Data heterogeneity* – occurs when data values differentiate and a conflict among multiple data sources (e.g. different ID, wrong spelling of the same values, missing data) arises

- *Structural heterogeneity* – includes differences with overlapping data values (e.g. aggregation differences [30] , missing items, homonyms [31] ,encoding mismatch [32] , types mismatch or generalization/specialization conflicts [33])

that has led to the current [53] most comprehensive form of four categories, evolved by dividing *Structural* category into *Language* and *Conceptual* (See Figure B.2 for details and examples).

The semantic heterogeneity has also become a target for applying the ontology-based integration approach. Explicit concept definitions and named relationships used by carefully designed ontology can bring powerful tools for handling the semantic mismatches.

2.2.6.2 Impedance Mismatch

Incompatibility in data integration of multiple, heterogeneous systems – also model incompatibility – is considered widely as a negative phenomenon. As a mere analogy to the electrical engineering, impedance mismatch refers to a set of conceptual issues when two incompatible data storing and manipulation models are being integrated. The impedance mismatch is a straightforward consequence of a *Data Independence* principle. The principle states that:

> “ (...)data, as a clue element of any computer system, should be independent of applications that retrieve and manipulate data. ”
>
> – *Data Independence Principle*

As a result there are several data independence levels:

- *Physical independence* – application is not aware of the physical details of data organization, which is managed by DBMS, only. Some of the details, however, (like indexes or file organization) are only available for *DataBase Administrator* (DBA).
- *Logical independence* – DBA can perform certain operations towards data (add/remove attributes, change user privileges, define views, program stored procedures, etc.)
- *Conceptual independence* – with the use of wrapper/mediator/views a database schema can be changed without or with only small changes to applications

This independence also forces the query language to be declarative as it has no notion of physical details of data organization (eg. presence of indexes). The most often case is using an object-oriented programming language to access the RDBMS. To name just a few of mismatching difficulties: declarative vs imperative model, syntax, typology, semantics, namespaces, target data nature of persistent (DBMS) vs volatile (programming language), etc.

The solution to this phenomenon is a wrapper/ODBC/JDBC based architecture at the data access, bottom layer of the application. The most common tools for impedance mismatch are the *Object Relational Mapping* (ORM) frameworks that provide patterns and design approaches for mapping rules for inter-model mismatches.

2.2.6.3 Transparency of Fragmentation/Portioning, Allocation and Replication

Considering a distributed environment/network of heterogeneous data processing elements in the form of data sources one has to carefully consider the way that the related data is being distributed across all the participants. The distributed database as a collection of interrelated

[30] E.g. the same population is divided differently (first-name, full-name etc.) by schema or sum/count added values differ

[31] E.g. refer to the same name referring to more than one concept.

[32] E.g. import or export of data to XML assumes different encoding types

[33] E.g. when single items in one schema are related to multiple items in another schema.

and shared data, physically distributed over network in some cases must consider three basic design aspects: *fragmentation, allocation and replication*.

Fragmentation (a.k.a partitioning) is essential for enhancing performance and provides simpler data management. As a way of dividing related data across data sources, fragmentation improves the system load balance by optimizing the hardware usage. There are two main factors that result in the system wide data fragmentation. Firstly, while considering large data volumes one has to consider the fact that data might be forced to be divided into a couple of sources in order to gain simple manageability and performance gain. Thus, data from one source has to be divided and moved to more stores.
Secondly, in the case of integration, two or more data sources can already store schema-related data, prior to the integration process. Therefore the integration has to consider that the global schema would have to merge the fragmented data stored on diverse data sources. There are three general types of data fragmentation: horizontal, vertical and mixed/hybrid.
Regardless of the two mentioned reasons for fragmentation the data fragmentation is always considered the same way. However, there must always be a possibility to define an operation that will reconstruct the entity from the fragments. The reconstruction operation for the horizontal fragmentation is `UNION`, while in the vertical case it is a `JOIN`-ing process. Each of the fragmentations is also required to satisfy completeness (see definition 2.7).

DEFINITION 2.7: *Fragmentation Completeness*

If an entity is decomposed into fragments $F_1, F_2, \ldots, F_n$, each data item that can be found in entity must appear in at least one fragment.

The horizontal fragmentation divides the data table horizontally, considering part of the entire table records as stored in a separate data source. The vertical fragmentation, on the other hand, considers dividing the table by splitting it, and considering some of the table records attributes as stored on the separate data source. Thus, data item is a tuple. In vertical fragmentation however, *Best Row Id* (BRI) must be repeated to allow reconstruction of each record. Thus the data item is an attribute. As a consequence, the third type of fragmentation is simply combination of horizontal and vertical types applied on different or the same groups of records of a target table for example in a RDBMS. The most obvious benefit of data fragmentation is that relation can be divided into a number of sub-relations which are then distributed. However, it also brings locality of reference or improved performance in terms of load balance the query processing costs.

Regarding the *data allocation* there can be discriminated the following data placement strategies:

- *Centralized* – Single DBMS stored at one site
- *Fragmented* – Data partitioned into disjoint fragments, with each fragment assigned to a different site
- *Completely Replicated* – Storing complete copy of the data at each site of an integrated grid
- *Partially Centralized* – Considers combination of fragmentation, replication and centralization

As for the *replication* in the integrated environment it has two sources. First, by integrating two databases that already share some subset of the same data. Secondly, there might be a need to copy some of the data for security or load balance reasons to more data storing sites. The presence of replication in many sites gives the following undeniable improvements:

- *Reliability* – In the case one site storing data fails, the replicated data can be acquired from the replicas.
- *Availability* – With intense system overload or security reasons of the primary data source one can forward queries to the replicas.
- *Performance* – Many queries can be redirected for the same data to separate data sources for load balancing.
- *Parallelism* – Parallelism is increased when read request is served.

The dominant characteristic of all of the above integrated data sources approaches is *transparency*. It states that the end user finds interactions with the integrated data source system as if there were no integration of structures, semantics, replications, fragmentations or data allocation schemas. Regardless of architecture the end user should not be obliged to know any of the system characteristics and nevertheless should be able to use them with full power of queries or API.

2.3 Data Stores - the Integration Targets

Classical integration solutions and approaches discussed in the previous sections have been designed to deal with multiple heterogeneous sources. Especially now, eclectic mixes of structural and model constructs across the modern integration systems have proliferated. In this section the author draws a sketch to only briefly classify the current state of the art in the field of storage sources as a target ingredient of data integration systems.

2.3.1 Database modelling - persistence

Right from the beginning of data integration the most common way to store and transfer data were *tabular data* files. This simplest way to store data involves numerous formats such as CSV (tab-delimited files-TSV, or any type of *SV delimiter), spreadsheets, fixed field formats, HTML tables, and SQL dumps. These formats enable displaying, or creating other formats. The file structure of an exemplary CSV file is based on rows, each containing information in the form of cells about a thing. Cells within the same column in a such format provide values for the same property of the things described by each row. Therefore such a file can store records with no structured relationships. A simpler incarnation of such a file is a file where each line contains unstructured data and for each line first two bytes define the format of the data stored in such a line. The undeniable advantage of flat file is that it take up less space than a structured file. The great problem, however, is that the application has to know, how to properly decode each and every line of the file. As a matter of fact, such solutions in the past were applied for the file systems (e.g. original Macintosh used flat file based *Macintosh File System* (MFS)) that has stored all the data in a single directory. At present some flat files can be found in the Unix-like operating systems namely the files such as `/etc/passwd` or `/etc/group`. As for the data transfer in recent years the flat file has consequently been replaced by the semi-structured files e.g. XML compressed files (according to *Efficient XML Interchange* (EXI) as a binary XML standard [54]).

2.3.2 Relational Model

The most accustomed and long time present data storing model is the relational model. The foundation of current relational concepts were formulated and proposed in [26, 27] based on the first-order predicate [34] logic (System-R [28, 29]). The Codd's proposition [26] involved logical

[34] Predicate is a statement that may be true or false depending on the values of its variables (i.e. predicate on X: $P : X \to \{true, false\}$). In terms of first order-logic exact semantic interpretation of an atomic formula and an atomic sentence says that the atomic formula consists of a predicate symbol applied to an appropriate number of terms.

concept of value-based data model. The data was supposed to be stored in tuples with named attributes while a set of tuples constituted relationships.

2.3.2.1 Relational Calculus and Relational Algebra

Relational calculus as a part of relational model [26] provides a declarative method for specifying database queries and is equivalent to the first-order logic. There are two components of relational calculus the *tuple relational calculus* and *domain relational calculus* [35] . The relational calculus refers to the quasi-natural language expressions used for non-procedural composing of SQL queries and statements [36]. Thus the relational calculus instead of queries (from relational algebra) would formulate a descriptive, declarative way. The *relational calculus* enabled defining the result of a query by describing its properties. Query in the relational calculus form would consist of two components: the target list and selection expression [56].

The relational calculus is logically equivalent to the *relational algebra* [13, 26], which is also part of the relational model, but provides a more procedural way of formulating queries based on the mathematical logic and set theory. It means that each algebraic expression can be formulated as an equivalent expression in the relational calculus. As stated by *Codd's Theorem* 2 – this is also true the opposite way. A database query can be formulated in one language if and only if it can be expressed in the other.

THEOREM 2: *Codd's Theorem*

Relational algebra and the domain-independent relational calculus queries are precisely equivalent in expressive power.

According to this theorem – query languages that are equivalent in expressive power to relational algebra are called *relationally complete*. So is relational calculus. However, there are exclusions from this general statement. It involves *aggregations*, *transitive closures* [37] or *SQL NULL*, nor any of the *three-valued logic* (3VL; with *true*, *false* and indeterminate 3rd value) they require.

Relational algebra and calculus constituted the basis of SQL. As a whole the relational model has never been implemented entirely according to its mathematical foundations. Relational model theory defines some basic terminology:

- *domain* – basic relational building block in the form of data type. Database interpretation is build-in types or user-defined ones.
- *attribute* – ordered set of attribute name and type name. Database interpretation is *column*.
- *tuple* – ordered pair of domain (data type) and value that forms an ordered set of attribute values (ir. valid value for the type of the attribute). Database interpretation is *row (a record)*.
- *relation* – set of tuples. Relation consists of *relation header* as a set of attributes, and *relation body* which is a set of n-tuples for an n-ary relation. Database interpretation is *table*.
- *relation variable* (a.k.a. *relation schema* or *relvar* [38]) – ordered pair of a domain and a name (relation header). Variable to which relation is assigned and the relation itself.

[35] Both are considered out of scope of this dissertation. Briefly speaking. Complete description of *tuple relational calculus*, that can be found in [26], deals with atomic values (atoms), operators, formulas and queries. Whereas the *domain relational calculus* details found in [55] discuss its declarative approach to query languages.

[36] Relational calculus has also played a significant role in design of declarative logic programming language – *Datalog* (syntactically a subset of Prolog). It is often used as a query language for deductive databases.

[37] First added as part of a declarative query in 1980 to Oracle Database with the use of `CONNECT BY... START WITH` statement.

[38] According to [57] introduced to distinguish the data that the relation is assigned to from the relation itself. The term is not widely accepted in enterprise solutions. There is a similar term of *base table* treating the concept the same

- *basic relation variables* – not derived from other relation variables. Database interpretation is *base table* created with the use of *CREATE TABLE* statement.
- *derived relation variables* (a.k.a *views*) – expression (using the operators of the relational algebra or the relational calculus) operating on one or more relations. When evaluated yields another relation. The view's expression can be named, and thus used as a variable name. Database interpretation is *view* created with the use of `CREATE VIEW` statement.

The basic rule for the relational model is the *Information Principle*:

INFOBOX 3: INFORMATION PRINCIPLE

All information is represented by data values in relations

and according to this principle, relational database becomes a set of *relvars*, and the result of every query is presented as a relation. Relvar is often related in context of *schemas*. A *relational schema* is a set of attributes tied with a set of constraints to define a set of relation. This often leads to another important aspect of relational data modelling which is the *data normalization* as the process of logical design.

2.3.2.2 Database Normalization

Normalization [39] of a database is a process of relation schemas with unwanted organization which are decomposed into smaller relational schemas with elimination of undesired characteristics. In general, such refactorisation of attributes and tables in a relational database ensure reduction of redundant data without losing information. This is in favour of data isolation so that CRUD operations can be conducted within only one table and with the use of foreign key propagated across the entire schema. The redundancy of data is for example a fact of existence of an attribute, in two or more tables – which causes inconsistency. This is due to possibility of the same attribute to become a different value in different places.

Functional Dependencies The starting point for the normalization process are the *functional dependencies* within relations. the functional dependency describe dependencies between attributes. This is a semantic predicate that must be *true* for arbitrary two tuples. Functional dependency is a characteristic of a realtion's schema, and not the relation's instance. Let us introduce some formal definitions:

as the variable in the programming language or *relation schema*. However, a *relation schema* is often considered as a *relvar* plus its constraints.

[39] From the point of view of integration, the normalization is a feature of only the relational-based solutions while the NoSQL solutions tend to marginalize its meaning.

Definition 2.8: *Functional Dependency*

For a given relation r with schema R. Let X, Y be subsets of attributes of R. In the schema R we say that Y is functionally dependent on X i.e.

$$X \to Y \Leftrightarrow \forall_{t_1, t_2} t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$$

where:
t_1, t_2 are tuples and
$t_i(A)$ states attribute's A value of t_i tuple

The relations that comply with the functional dependencies are called legal instances. Along inventing the relational model Edgar F. Codd has also introduced, at the same time the concept of normal forms – first [25], second and third normal [58] forms (1NF, 2NF, 3NF) and in 1974 the *Boyce-Codd Normal Form* (BCNF) [59]. Each consecutive NF must also conform to all predeceasing NFs. Now let us briefly discuss definitions of each of the normal forms.
We state that relation r from R schema is in:

- **1NF** – if all attributes values are atomic
- **2NF** – if none of its non-prime [40] attributes is dependent on any proper subset of any *candidate key* of the table
- **3NF** – all the attributes in a table are determined only by the *candidate keys* of that table and not by any non-prime attributes
- **BCNF** – if $X \to Y$ is a trivial functional dependency (i.e. $Y \subseteq X$) *OR* X is a *super key* [41] for R schema

The database is considered normalized if it is in 3NF – satisfy all its requirements. In this manner the database normalization improves queries performance at the price of schema complexity.

2.3.2.3 Object-Relational Database Model

So the main goal - of the ORDBMS - was to retain the declarative, predicate calculus [42]based query language from the RDBMSs', while adding the object concepts. Database schema and query language should also directly support classes, object behaviour with *Object IDs (OID)*, complex data with the user-defined-types and type inheritance. One of the most recognized projects that adheres to such object-relational model was Postgres developed at UC Berkeley. Many remains of the object-relational *evolution* have also become incorporated as a part of SQL:1999 [60] standard in the form of *structured user-defined types* (or briefly structured types) that are present for instance in Oracle database, IBM DB2, PostgreSQL or Microsoft SQL Server. This gives the user the ability to define custom data object types that can later be used just as in the case of native relational types like `CHAR` or `VARCHAR`. This means the object types can be either types of columns in relational table or types of variables. To explain the concept, let us discuss a tangible example of queries that is of object-oriented nature.

[40] Non-prime attribute of a relation (table) means that it is not a part of any candidate key of the relation
[41] Set of R schema attributes upon which all attributes of the schema are functionally dependent
[42] Predicate - (def.) is a verb template that describes a property of objects, or a relationship among objects represented by the variables. A predicate calculus is also called Logic Of Quantifiers

Listing 2.7: Declare *emp_type* object with methods [43] - PL/SQL style

```
CREATE TYPE emp_type AS OBJECT (
  eID           NUMBER,
  first_name    VARCHAR2(20),
  last_name     VARCHAR2(25),
  email         VARCHAR2(25),
  MAP MEMBER FUNCTION get_eID RETURN NUMBER,
  MEMBER PROCEDURE get_FullName ( SELF IN OUT NOCOPY emp_type ));
/
```

Listing 2.8: Define *emp_type* object with methods - PL/SQL style

```
CREATE TYPE BODY emp_type AS
  MAP MEMBER FUNCTION get_eID RETURN NUMBER IS
  BEGIN
    RETURN eID;
  END;
  MEMBER PROCEDURE get_FullName ( SELF IN OUT NOCOPY emp_type ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( first_name  ' '  last_name);
  END;
END;
/
```

In Listing 2.7 and 2.8 an exemplary object type has been prepared. Now one can create an arbitrary column of `emp_type` type or create a table of this type as displayed in Listing 2.9.

Listing 2.9: Define column and table of *emp_type* type

```
CREATE TABLE contacts (
  contact          emp_type,
  contact_date     DATE );

CREATE TABLE person_obj_table OF emp_type;
```

Then the following method execution would be possible:

Listing 2.10: Query column of *emp_type* type

```
SELECT c.contact.get_FullName() FROM contacts c;
```

Some syntactical differences between the dialects and the scope of object-oriented paradigm are comparable between the existing ORDBMS solutions.

2.3.3 Object-oriented Database Model

The object databases use the data model where information is represented in the form of objects in sense of modern object-oriented programming languages. The operations on data (in the form of objects) are executed with the use of integrated object-oriented programming language. This model thus overcomes the *impedance mismatch* while assuring the same model representation, and thus providing consistency between the data storage format and the data manipulation language. This object query language in also declarative [44] . Some undisputable advantages of ODBMSs are:

[43] According to Oracle PL/SQL Language `CREATE TYPE` statement

[44] The *Object Data Management Group* (ODMG) has even made some efforts in the past to standardize the *Object Query Language* (OQL).

- Expensive join operations are often needless, due to no need for tabular joining and no need for searching while following pointers are sufficient
- The same type definitions between the data store and the programming language
- Compared to RDBMSs, collecting massive amounts of information about a single item is easy as using pointers and changes from $O(n)$ to $O(1)$ (e.g. bank collects data about certain customer history including operations, debits etc.)
- Complex data are simply handled without need to mapping into relational rows and columns
- With the use of pointers many-to-many relation is simple for objects because pointers are linked to objects and establish relations

However, due to work suspension and withdraw of some standardisation participants and vendors in 2009 the model stagnated. Apart from enterprise ready solutions from *Versant* or *Gemstone*, there was a promising prototype of the Polish OODBMS project called *ODRA* (Java based) based on *Stack Based Approach* (SBA) [61] which however, unfortunately tends to be also deserted, so as its two remaining SBA implementations: LoXim [62] (C++ based) and PySBQL (Python implementation).

2.3.4 Column-oriented Relational Database Model (CORDB) – Relational Approach

Query processing in "classical", row-oriented (or record-oriented) database requires reads of entire rows (if no indexes). It is important that optimization requires storing of the data in the same area to minimize the *hard disk seek*, as the most expensive operation. This is known as a *Principle of locality* – i.e. data often accessed should be stored together. In context of row-oriented database, the principle implication is that the same record values will be frequently accessed together. Now to reuse the specific data with a relatively small answer duration time, and as the data is organized in records then the related records are put in the same hard disk block. Thus the number of blocks to be read is minimized. While this is good for the situations when one needs information about particular object, this is not efficient in the case of applying operations over multiple rows or data subset. In such a case a row-oriented approach would require entire data set seek.

Some workaround of this problem is the idea of *index* that stores column values and the pointers back to original *best row id* (BRI). By storing only the dedicated value from a row, indexes generally are much smaller than the entire table size, and thus scanning times with use of the indexes are greatly reduced. On the other hand, indexes bring an overhead to the system especially when considering new data updates. In such cases, the *non-clustered indexes* needs to be updated towards new, bigger data stored in tables. Some more in-depth level of index optimization are the *clustered indexes*. Such a cluster index – with contrast to a non-clustered index – changes physical data records storage order, to match the order of index. This way the data can be retrieved much faster in the cases where we need to access the data sequentially according to the same or reverse order as the one imposed by the index. However, due to this assumption, there can be only one clustered index per table.

A more general answer to this class of access optimization problems is a dedicated column-oriented data model. In this model each column is closely similar to the idea of an index in a row-oriented model. It redesigns the way of organizing data stored in hard disk blocks by serializing all of the column values together. Moreover, if the next column would fitted the same block, it would also be stored next to the first one, right in the same manner. What discriminates this model from the row-oriented model with index on every column is the mapping. In contrast to row-oriented model, where the rowID is the *best row id* mapped to the index data – in the column-oriented model – the BRI is the data that is mapped to rowIDs. Thus one value is stored

in the column-oriented model only once[45] . Therefore, searching for all its occurrences boils down to a single retrieving operation. Moreover, aggregating operations (counting, summing, math operations) over a set of data can gain optimization improvements through this model.

However, the efficiency drops, in the case of object (record) retrieving queries that would be much slower, as they would require to scan over couple columns that are stored separately. However, complete record operations are relatively uncommon. Mostly it is required to gather only a subset of records attributes (columns values). The column-oriented model is also efficient in the case of operating on the sparse data where multiple columns presence is optional due to the attribute values belonging to the same column stored contiguously, compressed, and densely packed. In general we can state that the more rows in table and the bigger size of a row is – the more we gain with the column-oriented model.

$$\frac{number_of_rows \times size_of_row}{size_of_db_page^{46}} \approx number_of_IO_operations \tag{2.5}$$

The column-oriented model has proven its real-world good performance [63] and fitness for the purposes such as OLAP analysis, reporting or *Decision Support Systems* (DSS).

The most importantly, it is not possible to move the column-oriented benefits to the row-oriented model in a straightforward manner by simply using vertically partitioning of the schema, or by using the index towards every column to be accessed independently. To gain the column-oriented advantages in the row-oriented model it requires considerable changes of the storage mechanisms and query executor engine (i.e. vectorized query processing, or compression) can bring even more performance from the column-oriented model [64].

While the row-oriented model is still the better choice for the OLTP operational systems, however the OLAP analytical approach for aggregating and reports can profit more from applying the column-oriented model in terms of performance and data storage space optimization. Therefore, the column-oriented model is most often used in the *data warehousing* implementations. The most profound representatives are: *Sybase IQ, Vertica, ParAccel* and *Infobright*.

2.3.5 NoSQL – Distributed Storage Services

The NoSQL stands for *Not Only SQL*, which is often mistakingly referred to as *No SQL* - the difference is enormous. Contrary to the misconceptions caused by its name, it does not prohibit SQL in general, but labels the *"Not ONLY SQL"* approach. Because there is no strong NoSQL definition, to become more specific, the author has developed the following formalization.

DEFINITION 2.9: *NoSQL databases*

NoSQL databases - is a class of DBMSs that had to give up the set of the tight relational rules that become obstacles to meet the challenges, and were sufficient to fulfil the requirements of concrete application while preserving the rest of the principles and regulations.

Below there are some principles that have driven the NoSQL emergence.

- Availability comes first
- Fixed database and table schema are not required
- Simplicity of design
- Strict rules, that govern transactions in relational databases, are not required

[45] Consecutive duplicates within a single column may be automatically removed or compressed efficiently.
[46] E.g 4KB, 8KB, 16KB, etc.

- Costly joining operations are undesired
- Store denormalized data due to join elimination
- Horizontal scaling should be simple
- Designed to run on large clusters

On the whole, it can be brought to a point when we consider the NoSQL as a flexible schema solution that is quicker, and thus cheaper to setup, and provides massive scalability at the cost of relaxed consistency due to high performance and availability. However, one has to be warned, that the relaxed consistency means fewer guaranties, and a lack of declarative language simply implies the need of more code, and thus more error prone development.

2.3.5.1 NoSQL Motivation

Considering a general set of requirements that defines a RDBMS, in some applications, some of those requirements are not always necessary, and what is more, it sometimes might be advantageous not to have it supplied. This is where NoSQL comes in, assuring only a subset of requirements that a RDBMS would have to provide. Due to the networking nature of the modern computer systems NoSQL databases has become a convenient way to work on the distributed nodes across network. Such nodes did not aim at rich query abilities, as it was not considered as important as the improved availability at the cost of consistency. However, most of the NoSQL implementations provide only the "eventual consistency" model [47] . The model states that after some data manipulating operation, it will eventually become consistent after some time if there is be no further interaction with the manipulated data for a sufficient period of time.

According to *CAP Theorem*, high efficiency of read/write access – i.e. availability – in a distributed data store, implies the disadvantage of sacrificing the consistency. This is not only in order to high availability, by also partition tolerance.

2.3.5.2 NoSQL Models

There are five main incarnations of NoSQL store models that are being presently widely used.

Key-Value This model has been designed for more OLTP kind of operations. It involves small operations over a small number or single records in a massive database. The data model is very simple as it involves the *keys* and *values* pairs and support *Insert, Fetch, Update and Delete*. The result is a very fast solution that can assure efficiency, scalability and fault-tolerance. This is mainly due to implementation based on records distributed to nodes based on key (e.g. hash value over the key), supporting record replication across the multiple nodes with a single-record (i.e. one operation; no record groups) transactions and eventual consistency. One has to be aware that in the key-value store the data is considered to be inherently opaque (i.e. form of a BLOB) to the database and the data is treated as a single opaque collection which may have different fields for every record.

Column The key-value data model with its operations is sometimes not enough and therefore some key-value stores introduced a concept of *column* within the *value* part of the model. So that the value enables more structure than just a simple BLOB of bits. The resulting *Column* model resembles the embedded key-value store with a non-uniform, strict structure of the columns. This model also enables fetching operation that works on a range of keys in contrast to

[47] Often classified as providing *Basically Available, Soft state, Eventual consistency* (BASE) semantics, in contrast to the traditional ACID.

a single key in the regular key-value store. To contrast the column model to RDBs, a counterpart of schema from RDBMSs in the *Column* model is a *Keyspace*. Thus in the most frequent cases the column based applications have only one *keyspace* - which is the most general abstract container of this model. On the other end, the most basic element and low level object that the *keyspace* contains is a *column*. As the smallest increment of data, the column is represented by a *tuple* (a key-value pair) of three elements `<name, value, timestamp>`. The timestamp value is important due the to actual lack of consistency and distribution of data nodes. By dint of the timestamp one can check if the data stored in a replicated/backup node is up-to-date. A *Super-Column*, on the other hand, is a tuple with a name and a map of unbound number of columns. The exemplary column and supercolumn are depicted in Listings 2.112.12

Listing 2.11: Column

```
1 {      name  :"emailAddress",
2         value :"john@example.com",
3  timestamp:  12345
4 }
```

Listing 2.12: Super-Column

```
1 {    name:  "homeAddress",
2      value:  { // map of unlimited number of columns
3      //|  KEY   |              COLUMN VALUE                    |
4           street :{name:"street",value:"Balladyny",timestamp:12345},
5           city:   {name:"city",   value:"Lublin",    timestamp:12345},
6           zip:    {name:"zip",    value:"20-601",    timestamp:12345},
7      }
8 }
```

Again each column and super-column can become part of a *Super-/ColumFamily*. Column-Family can be compared to *rows* from the entity-relationship model. ColumnFamily contains a number of columns or super-columns, however this is where analogy ends as there is no schema enforced at this level and its *rows* do not have a predefined list of Columns that they contain. Moreover row can change in structure between consecutive updates.

Listing 2.13: ColumFamily - simplified notation - i.e. no timestamps and column/super-column names removed

```
1  UserProfile = { // ColumnFamily
2      JohnDoe: {    // Key to this Row inside the ColumnFamily
3          // infinite nr of columns in this row
4          username: "John Doe",
5              email: "john@example.com",
6            gender: "male"
7      }, // end row
8      JackSmith: {    // Key to another Row in the ColumnFamily
9          // next infinite nr of columns in this row
10         username: "Jack Smith",
11             email: "jack@example.com",
12             phone: "+48 555 444 444",
13               age: "33"
14     }, ...
15 }
```

Therefore, ColumnFamily is more like a *HashMap/dictionary* or a *associative array*. At this level we move on to a *Keyspace* as a grouping container for ColumnFamilies and a single and most general container for application's data. The column model, in the case of distributed data stores,

makes the row handling all pushed towards programmers. The most prominent representatives of this Column based model are Google BigTable, Cassandra and HBase, but there are also Amazon Dynamo, Voldemort and more.

It must be clearly stated that *column stores* or more often called *columnar databases* in another sense are a way of organizing RDBMSs data in columns instead of rows for higher performance in certain applications. Owing to this, data can be highly compressed which allows aggregating the functions – like MIN, MAX, SUM, COUNT and AVG for rapid performance boost. The second advantage of columnar arrangement is that it is self indexing and uses less space than regular RDBMS with the same data.

CODBMS vs NoSQL column model Here the author is obliged to explain the high level discrimination between the *Column-oriented DBMSs* and the *NoSQL column model*. To name just a few representatives of both groups:

- **CODBMS** (relational) – Sybase IQ, Vertica, C-Store, MonetDB, VectorWise, ParAccel, and Infobright use column based storage and access [48]
- **NoSQL column model** (non-relational) – BigTable, HBase, Cassandra and Hypertable are able to store and access column families separately

There is no widely accepted terminology that could allow to divide and classify existing solutions into these two groups. However, there are some important design differences that can help to provide a reasonable taxonomy. The main differences are depicted in Figure 2.6. Additionally,

Characteristics	Non-Relational Column Sore	Relational CODBMS
Data Model	• sparse, distributed, persistent multi-dimensional sorted map (`Row-name`, `Column-name`, `Timestamp`) → `value` in database i.e. non-relational model	• Traditional relational model
Independence of Columns	• Store parts of a data entity / "row" in separate column-families and column-families are accessed separately • Not all parts of row are picked up in a single I/O operation from storage → optimization in case when only subset of a row is relevant for particular query • Column-families may consists of many columns while its component **sub-columns can not be accessed independently**	• **Every** column is stored separately in traditional relational database table • Best suited for queries requesting only subset of table attributes
Interface	• As a part of NoSQL no SQL interface	• Supports standard SQL interfaces

Workload Optimizations	• More diverse set of applications e.g. handle higher update rate
	• Struggle with aggregation-heavy workloads
	• Best for **individual row queries** due to:
	  • possible column-family placement of co-accessed attributes (save on seek cost compare to separate columns as in relational CODBMS)
	  • storage layer implementation (see characteristic below)
		• Best suited for read and analytical workloads with fast load times – **aggregations** over large number of similar items scanning many rows in a single query
		• Low update rate
		• Sourcing from single-column row representation
		• Appliance: *data warehouses* due to bulk-loads, read queries and rare updates and *ad hoc* inquiry systems (i.e. CRM, catalogues)

Storage Layer	• Sparse data model implies:
	  • different rows can have very different set of columns defined – thus, explicit storing (`row-name/column-name, value`) pairs for element within each column/column-family
	  • No NULL filling due to sparse nature
		• All **values** from a single column are **stored consecutively** without notion of row or column
		• Above implies that undefined column values for a row must be **NULL** to match up values based on their positions in corresponding lists representation of columns
		• Typical take less storage space for structured data (and not storing column/row names)
		• Optimized for column operations: *read column element and apply operation* (i.e. predicate evaluation or aggregation)

Figure 2.6: Taxonomy of heterogeneity

one of the creators of *Vertica* (developed from *C-Store* [65], and also actively involved with H-Store) has highlighted the following distinction between the two models:

[48] Those are strictly column-oriented model representatives. However, this list can be larger as there are more solutions of hybrid – column and row paradigm elements – such as: IBM DB2, Amazon Redshift or Google BigQuery, to name just a few best known.

> (...)(*non-relational model solutions*) are really row stores. I.e. they store a column family in a row-by-row fashion. Effectively, they are using materialized views for column families and storing materialized views row-by-row. Systems in (*CODBMS*) group have a sophisticated column-oriented optimizer – no such thing exists for (*non-relational group*) (...).
>
> – *prof. Michael Stonebraker*

On the whole the the CODBMS are best suited for OLAP kind of applications requiring complex queries towards large volumes of data (i.e TB, PB, etc.). While in the case of non-relational column-model it can deal easily with many simple and row oriented queries.

Document This category is one of the most popular among the NoSQL world possibly due to being designed for managing of a semi-structured data. The document-oriented model may seem to be a straightforward inherent subclass of a key-value store concept. However, the difference is the data processing. As already mentioned, with the key-value stores, the data is opaque to the database. Within the document stores, however, the *value* is a data *document* containing its internal structure (i.e. metadata) which is represented in the user-readable form (XML, JSON, Yaml, semi-structured formats, or even binary formats like BSON), that is understandable and used by the database engine. This way the metadata based document structure can be a subject for automation while staying direct towards the actual data instead of being tied by its schema. In general, we can call this model a *key-document model* that – in contrast to the key-based *Fetch* from the key-value model – additionally enables fetching on document contents. As a sub-class of key-value store, in general the document stores also differ from RDBs. The main difference lies in typing. The RDBs store strongly typed data, while the type is introduced in process of schema creation and is limited to a given set of types. Once created one has to follow the given type schema and changes are considered uneasy and complicated. In contrast, the document-oriented stores obtain type information from the data itself as it is kept together with the data. Data formats are not predefined in the document case. Moreover, each instance can change its type set and can vary compared to any other instance of the data. Thus, one can benefit from flexibility to change the metamodel and provide optional values, which is especially very useful while mapping onto the existing concepts of object-oriented programming (OOP). This is why the *impedance mismatch* problem of RDBMSs does not exist in a document-oriented system or in general, in the NoSQL systems.

The most straightforward and obvious implementation of the document-oriented database are the XML databases that use XML to extract metadata information from it and sometimes even store the actual data. However, the most popular document-oriented DBMS is MongoDB.

Listing 2.14: Raw XML based document

```
1 <address>
2         <fname>John</fname>
3         <lname>Doe</lname>
4         <street>Kwiatowa</street>
5         <city>Lublin</city>
6         <zip>20-601</zip>
7         <country>POLAND</country>
8  </contact>
```

Listing 2.15: JSON-based document; MongoDB style

```
1  {
2       _id: <ObjectID>,
3     fname: "John",
4     lname: "Doe",
5       //embedded sub-document
6    adress: {
7        street: "ul. Kwiatowa",
8          city: "Lublin"
9    }
10 }
```

Listing 2.16: Metadata document for page node

```
{
    _id: ObjectId(...),
    nonce: ObjectId(...),
    metadata: {
           type: "basic-page",
        section: "my-photos",
           slug: "about",
          title: "About Us",
        created: ISODate(...),
         author: { _id: ObjectId(...),
                   fname: "Michael" },
           tags: [ ... ],
         detail: { text: '# About Us\n", lastedited: "" }
    }
}
```

Overall, there are multiple implementations of this paradigm like *MongoDB, CouchDB, Redis, MUMPS, RavenDB* and many more.

Graph This model was designed for storing and querying over very large graphs. The model involves two kinds of objects. A *Node* and an *Egde* (which is placed between two nodes). The node contains *properties* (like node ID) and edges have *labels/roles*. The node properties are again key-value pairs. The operations or querying languages available for graph databases are not standardized, therefore there are single step-queries (e.g. in the social graph ask for all friends of a person-node), path expressions [49] , but also a full recursion [50] . The most significant of the graph systems is Neo4j, but again there are many more like *FlockDB, Pregel*, etc.

Resource Description Framework (RDF) *RDF* is based on the relations between objects, so one can find resemblance to nodes and edges from the graph model. The RDF is a family of World Wide Web Consortium (W3C) specifications [66, 67] aimed at representing a data model as metadata. Just as in the case of class diagrams it is based upon the idea to make *statements* about *resources* in the form of *subject–predicate–object expressions* (a.k.a *triple stores*). This similarity and the triples idea make it easy to map the RDF *triple stores* to a graph database. One of the most prominent RDF query languages is *Protocol and RDF Query Language* (SPARQL), that is a semantic query language for the RDF databases used for data manipulations.

MapReduce This model should be considered as the one of couple of systems [51] designed for more OLAP and analytical kind of operations that involve scanning most of large amounts of data to do complex analysis. It originally comes from Google. Its distinctive characteristic is that there is no data model at all. The data is stored in files e.g. *Google File System* (GFS) or for the Hadoop open implementation version – HDFS. The operations, in case of MapReduce model – are limited to *Map, Reduce, Reader, Writer* functions that will be extensively discussed just as the entire Hadoop ecosystem in Section 2.4.3.3.

As the regular DBMS the MapReduce is sometimes used for views and queries as it is with *CouchDB*.

Multi-Model In contrast to most of DBMSs, a new *Multi-Model* database design approach has been introduced. Its basic feature is that it can use multiple models against a single, integrated

[49] Following edges and nodes to find e.g. female friends of friends for a given person-node.
[50] Traversing in-depth the graph for a given depth.
[51] See subsection 2.4.3.3 for examples with details.

backend. As already discussed in Section 2.2.3, challenging enterprise application often requires multiple data models to be supported and therefore solving this problem, can be done by adopting strategy pattern of *polyglot persistence*. In a straightforward manner this solution has two major drawbacks. It increases the operational complexity [52] and it does not guarantee data consistency across the integrated data sources [53]. These problems are target for the multi-model solutions that aim at reduction of the operational complexity with provision of single data store [54] [68].

The examples of this model are: *FoundationDb* (that not only supports multiple layers of NoSQL solutions but it also supports ACID), *Aerospike, OrientDB, ArangoDB, Couchbase Server 2.0, Riak* (as a key-value store it accepts JSON as a value) or *NuoDB*. Most of those solutions like the OrientDB, ArangoDB or FundationDB supports SQL, key-value/document/graph/object stores and ACID functionality; in some cases even distribution.
Moreover, some of the general-purpose databases also have multi-model options, like the *Oracle MySQL 5.6* that support both SQL and key-value access with the *Memcached* API or *PostgreSQL h-store* that can store key-value pairs within a PostgreSQL data field, thus enabling schema-less queries against data in PostgreSQL.

Some of those solutions often belong to a class of modern RDBMSs that provide scalable performance of NoSQL for the OLTP procedures while maintaining ACID guarantees. For some time those products have been named *NewSQL*.

2.3.6 NewSQL

Despite of fact that NoSQL trend evolved from the relational model as a response to new massive amounts of data, the well established and reliable relational solutions were not able to face new challenges. This was due to conceptual model issues that the traditional relational solutions suffered from design. However, the NoSQL movement is based on highly specialized solutions that complement the transaction-ready solutions rather than replace them. This situation was noted by new DB vendors that have presented brand new approach to data storage that intended to merge the best from both worlds. This involved the attempt to improve the relational model for distributed systems and vertical scalability.

The current situation involves large scale systems, with huge and growing data sets (like Facebook 9M messages/hour or Twitter 50M messages per day), often generated in an automated way by software and devices. Modern systems require also high concurrency requirements data model that requires some relations and transactional integrity. As a result, the architectural trends changed towards moving traditional dedicated databases into cloud due to consistency, transaction handling, storage optimization and scalability purposes. The NoSQL as a new kind of non-relational database products have rejected a flexible table schema and join operations. Its goal was to meet the scalability requirements for distributed architectures while presenting none schema data management requirements. The CAP and ACID were replaced rather by the BASE architectural trend. NoSQL provided also horizontal scaling with the use of custom APIs. This forced the additional application level sharding and logic to provide functionalities previously present in classical, relational solutions. Moving the logic to applications has risen a complexity and support cost of such systems. This was the reason that the new class of solutions, called *NewSQL* 2.10, has emerged.

[52] Each data storage mechanism introduces a new interface to be learned.

[53] The purpose of this thesis is to provide a solution that would eliminate this disadvantages while still providing the polyglot persistence. See following chapters for details.

[54] However, single data store requires data migration which is not always possible or affordable. The thesis provides here a meta solution that eliminates this problems.

DEFINITION 2.10: *NewSQL*

The NewSQL term refers to new products (rather than SQL itself) that:

- deliver scalability, flexibility and transactions over distributed architectures [a] , while retaining the SQL support queries and ACID, or
- improve performance in terms of vertical scalability, if that is good enough, horizontal scalability is no longer a necessity.

[a] i.e. what the NoSQL complements towards RDBMSs

The goal of the *NewSQL* concept is to provide solutions with the SQL-based interface with the ACID support but still with non-locking concurrency control, high per-node performance with scalable, shared nothing architecture .

> I would define a NewSQL DBMS as one having the following 5 characteristics:
>
> 1. SQL as the primary mechanism for application interaction
> 2. ACID support for transactions
> 3. A non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thereby cause them to stall.
> 4. An architecture providing much higher per-node performance than the available from the traditional "elephants"
> 5. A scale-out, shared-nothing architecture, capable of running on a large number of nodes without bottlenecking
>
> *– prof. Michael Stonebraker (see [69])*

The shared nothing architecture eliminated the single point of failure as each node is independent and self-sufficient. There is no shared disk nor memory that enables infinite scaling and data partitioning.

The reasons for NewSQL approach was due to a couple of aspects. The OLTP solutions as designed and optimized towards computers in 1970's include disk-resilient B-trees, heap files, locking-based concurrency control has changed a little compared to the changes that took place on the ground of hardware. It moved to an extent where in some cases the OLTP database can fit into memory, thus the transactions can take less than milliseconds. As shown in [70] performance exploration of modern conventional databases transaction processing (TP) components there are many overheads – see Figure 2.7. The test has proven that eliminating the overheads has increased the number of transactions per second (TPS) from 640 up to 12 700 TPS with the single-threaded, lock-free query processing kernel. The most overhead was caused by *buffer management* that has involved buffer pool for accessing data stored on the fixed-size disk pages – providing level of indirection while accessing each record which must be located on those pages with identified field boundaries. What is more, the buffer pool determines which set of disk pages is cached in memory at a given time. One can eliminate this aspect of TP if data is held in memory for the maximum throughput. In the cases data does not fit (due to size) the memory, one should consider *Anti-Caching* [71] that enables better main memory management and does not require data conversion between the main memory and the disk format.
The second most significant issue is the records write-ahead *logging* and change tracking. The duplication of each write , i.e. the database and the log writes, that are stored at disk for transaction durability provide the second biggest overhead. However, in the case of negligible recoverability or its external assurance (e.g. automatic / external / versioned replicas in the intra- and inter-cluster high availability nodes) one can bring approx. 20% TPC gains while stripping

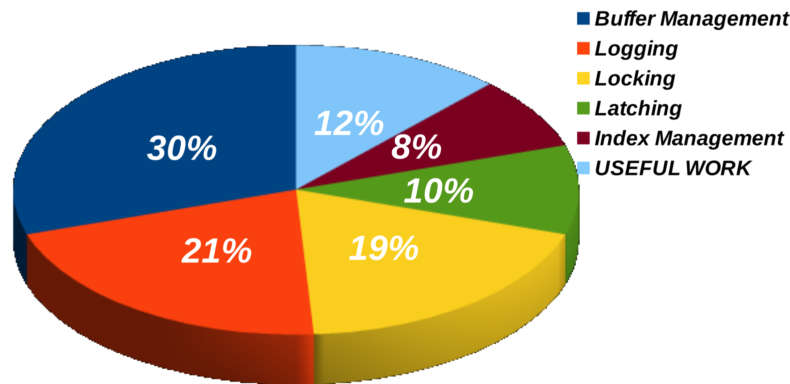


Figure 2.7: Traditional OLTP overheads.

this functionality.
Next is the *locking* – in terms of traditional two-phase locking governed by Lock Manager entity it requires setting a lock on the lock table.
In the case of arbitrary data access optimization B-trees, hash tables and any other disk-based index structures require also additional CPU and I/O. In the case of in-memory database this can become a significant overhead with classical B-trees and might require more cache-conscious index e.g.[72].
The multi-threaded nature is assured with the use of short duration *latches*, while updating the database shared structures (e.g. B-trees, lock tables, resource tables, etc.). This is required during every multi-threaded database access. This overhead can also be considered redundant, and eliminated, by using single-threaded implementation with autonomous operating partitions. One can achieve this way database parallelism due to distribution of single-threaded, autonomous execution nodes among cluster wide servers' CPU cores.
This show how potential is in careful elimination of such overheads.

> “ (...) removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance. ”
>
> *– VoltDB white paper (see [73])*

Those overhead components, originally aimed at assuring the *data integrity*, however currently they prevent the modern systems based on traditional databases from scaling, thus facing modern workload and data volume requirements. The traditional *scale-up* approach is costly in terms of hardware and software, but what is more important it generates additional complexity, thus the maintenance costs also increase.

Therefore, the modern – NewSQL – approach considers the *scale-out*, shared-nothing architectures with close to linear scalability. Moreover, *NewSQL* solutions tend to apply the well tested and used for a long time architectures and provide their an in-depth tuning moving towards modern *scale-out* requirements. This often involves stored procedures interfacing for transaction handling, using main memory based architectures with automatic partitioning across the shared nothing server cluster.

The NewSQL solutions can be classified under three main categories:

- *New Approach*: VoltDb, Clustrix, NuoDB, etc.
- *New Storage Engine*: TokuDB, ScaleDB, etc.
- *Transparent Clustering*: ScaleBase, dbShards, etc.

The first one to deliver the NewSQL oriented solution was Google. The company answered to SQL demand (not answered by NoSQL) [55] with *Spanner* and *F1* projects that enabled globally-distributed transactions, thus pushed SQL further than ever. However, recently new vendors introduced interesting solutions. This involved VoltDB [56] , an in-memory DB, aiming for speed and make SQL semantics work over streaming applications. Similarly to *SparkSQL* (see section 2.4.3.3) and its streaming module that allows to stream data through and write SQL queries that evaluate over a chunk of data that comes in. VoltDB uses stored procedures interface with asynchronous/synchronous execution with data access serialization and horizontal partitioning. It also enables multi-master replications with *K-safety* [57] . This means that the appropriate number of partitions out of the cluster will be assigned as duplicates, and ensure that the duplicates are kept on separate nodes of the cluster in order to assure given K value.
Another project is *NuoDB* that uses the distributed ACID transactions with D supported as a key-value store. NuoDB is a *CP* class system that needs majority of nodes to work. This is the same as another *CP*-class system named *Clusterix*. Its goal is to provide real-time analytics with distributed SQL in terms of performance for better resiliency in. A great example of evolution of the "legacy-SQL" databases is *MariaDB Galera Cluster* as a fork of MySQL. The vendor here assumed the evolutionary approach to re-engineer and extend the existing solution to be more distributed and more like the NoSQL solutions. Additionally, joint effort initiative of Facebook, Google, LinkedIn and Twitter engineers called *WebScaleSQL.org* consortium tries to coordinate efforts to make SQL more scalable. The new approach has been part of the *TokuDB* that aims at high performance storage engine for the use with *MariaDB* or *MySQL*. It uses *fractal tree index* [58] for performance while still complying ACID and *Multiversion Concurrency Control* (MCC or MVCC) [59] , a concurrency control method to provide concurrent access to the database. The others are: the GenieDB and ScaleBase (as globally distributed MySQL as a service DBMSs), ScaleDB (serves streaming inserts and analytics with SQL) and TransLattice (globally distributed transactional SQL-based on Postgres). In general the solutions are based on taking the existing technology and adding underneath a distribution or on top of it a better management.

2.3.7 Big Data - all or nothing

However NoSQL, is simply a part of a bigger trend that has emerged. Due to the natural growth of digitized global information, the data to be stored and analysed has become unmaintainable with the traditional approach. According to the *Science* article [74], the world per-capita capacity to store information has doubled every 40 months since 1980. Every day new 2.5 exabytes [75] (2.5 quintillion or 2.5×10^{18} bytes) of data are being created. What is more, the network throughput capacity also increased from 281 (1986) and 471 (1993) petabytes to 2.5 (2000) and 65 exabytes in 2007. As for 2014 only the mobile global traffic reached 2.5 exabytes per month. This means 69 percent growth from only 2013 with 1.5 exabytes per month. With forecasts of 24.3 exabytes per month [76], considering only the mobile traffic with *Compound Annual Growth*

[55] The BigTable proved good for storage but insufficient for SQL.

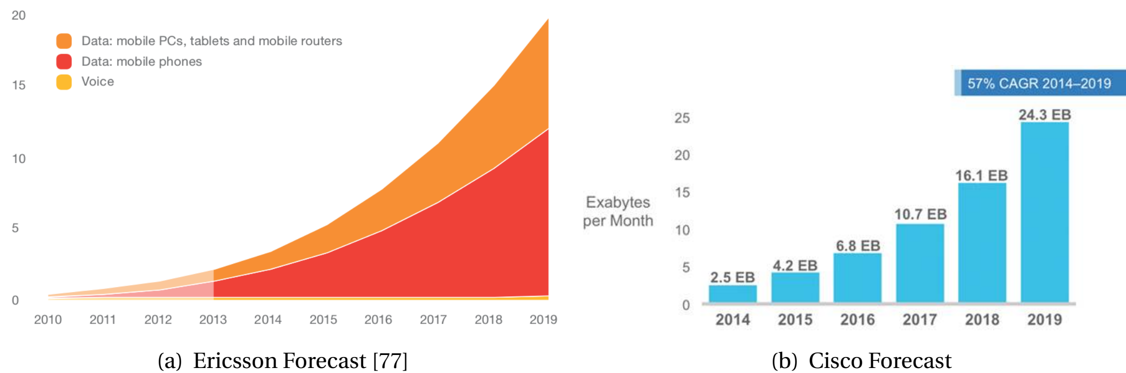
[56] From M. Stonebraker the creator of Postgres

[57] Property of a cluster that defines number of nodes that can fail while the database continues to run normally. Most often this requires calculating the maximum number of unique partitions that can be created with the given number of nodes, partitions per node, and the desired K-safety value. If more than K nodes fails, the system is no longer K-safe and turns off to prevent inconsistencies. E.g if $K = 1$ then half of partition will be assigned as duplicates of the second half, if $K = 2$ then 2/3rds of the cluster's partitions will become copies of the remaining 1/3rd

[58] A generalization of *binary search tree* (BST)(just as B-tree) tree data structure with the sorted data, with access times the same as B-trees, but with insertions and deletions asymptotically faster than the B-tree. This is due to more than two children and that each node has buffer allowing to store update operations (insertions, deletions) in intermediate locations, thus scheduling disk writes.

[59] The simplest method is *locking* which is when readers must wait until the reader stop writing. However, this makes the system slow. With MVCC each user sees a snapshot of the database that was consistent at that particular time. Any changes made by the writer are made available to the readers only after writing transaction which is marked *complete*.

Rate (CAGR) 2014-2019 of 57%, brings the idea of Big Data to tangible numbers. The Big Data



(a) Ericsson Forecast [77]

(b) Cisco Forecast

Figure 2.8: Monthly Mobile Data Traffic Forecasts by 2019

definition has been coined in numerous ways, [78], or [79]. However, the problem however, was discussed much earlier in technical report [80](2001) from META Group (later acquired by Gartner). It introduced the *3V principle* describing data growth challenges in 3D manner.

INFOBOX 4: *3D BigData model* (3V PRINCIPLE)

BigData's 3D nature is composed of the following dimensions:

- **Volume** - amount of the data. It is important in the context of BigData that we deal with characteristic of the 'Big' data
- **Velocity** - data generation time. The speed of data generation and processing to meet the demands and challenges put ahead due to growth and developement
- **Variety** - quantity of data types and sources. Heterogeneous data sources, their types and particularities are getting more diverse as BigData is being developed and more demanding. Moreover BigData extends beyond structured data, including all unstructured data varieties text, audio, video, click streams, log files and more.

Veracity and *Value* are sometimes added. Veracity depicts the data cleanliness or quality and how precise it get so that the end user can be sure the data is accurate. It answers the question if the data is trustworthy. It must consider the aspects of: duplication, ambiguities, latency, inconsistencies and eventual approximations. On the other hand, the *Value* takes into account the business value of the collected data. Therefore, one must consider what benefits will be provided due to cost generated while collecting the data. In this manner one way of accessing the data is to integrate it in a structured way of purpose-built format (like files or folders in the structured data warehouse) or in the form of *data lake* where it is copied as-is, in its native data format and stored for further analysis. The data lake idea is driven by eliminating the cost of data ingestion (like transformation), increase agility and accessibility. However, the noticeable downsides are a lack of semantic consistency, governed metadata, and need for additional data manipulation prior to actual data analysis. The true beginning of the big data was in 2012 when the U.S. White House introduced the "Big Data Initiative" with $200 million intended for multiple research and government agencies like AMPLab at UC Berkeley, DARPA, Department of Energy, National Science Foundation (NSF) etc. This was also when the definition for BigData was widely commercially adopted [81] and become re-coined by Gartner as presented in Definition 2.11

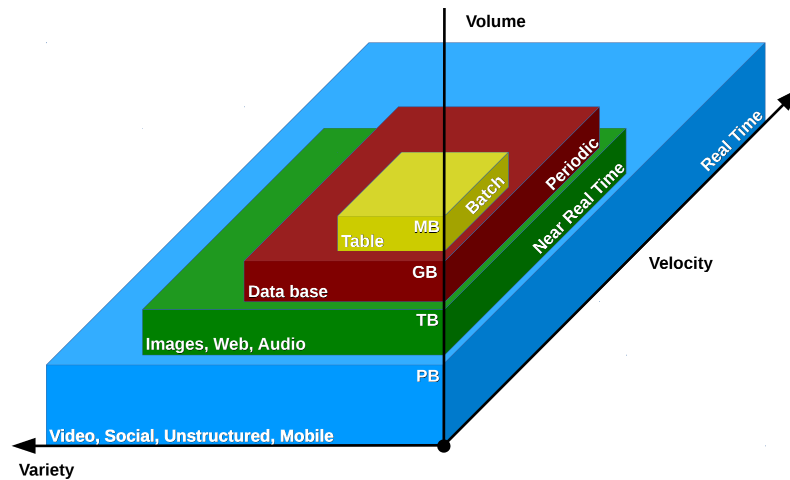


Figure 2.9: BigData: Expanding 3D data space

DEFINITION 2.11: *BigData* [a]

Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.

As of the present (2015), the most recent definition, that aimed at becoming the most consensual one, recalls BigData as follows:

> Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value
>
> – *See [83]*

Traditional data processing [60] applications – in the BigData era – proved to be inadequate and insufficient. Using the classical solutions towards BigData resulted in difficulties with data analysis and retaining reasonable costs at the same time. The issues with BigData support by traditional solutions, forced development of dedicated, new solutions broadening the entire spectrum and brought new, cutting edge ideas.

Due to rapid technology changes BigData is constantly changing, however, it is always all about unveiling large hidden values from large datasets. What is especially important is that regardless of what set of techniques or technologies is included in BigData term, it will always require new ways of integration for heterogeneous, diverse, complex and massive scale data that can bring agility and work unconditioned to the data sizes.

The goal of this dissertation is to introduce such a new way of data integration that will be elaborated in the following chapters.

2.3.8 After SQL Era

Back in the past the only methods to store data were flat files and the SQL solutions. The Internet made everyone to rethink persistence in a way that brought the NoSQL and many of its essentially new ideas. And finally in recent years, there has been recap that SQL was actually very

[60] Like the basic: validation, sorting, summarizing, aggregating, analysis for data interpretation, reporting computed information or classification that separates data into various categories.

valuable and should not be abandoned while following fashion for NoSQL. Therefore, return of relational model led to reconsidering the approach to scalability, availability, etc. There are some interesting new areas developing recently in the area of data approaching. However, SQL is sure to be always present. The NoSQL revolution expanded the possible options and become important part of the persistence tools set.

Graphs The graph data representation model has already been present for some time with some promising implementations – e.g. Neo4J. However, graphs are often misconcepted as an object models. To determine a model as a graph model one could think of trying to answer the ACID compliance question – i.e. would the standard graph traversal algorithms make sense (e.g. find the connected components with the shortest path, minimum spanning tree, find cliques in a graph, etc.)? Moreover the model is probably more like a relational model. On the other hand, if a relational model requires more and more joining the graph model should be considered useful.
In modern system a sharding problem is a very important issue. In terms of graph, finding the place where we put the cutting line, and answering questions such as how graph can be cut into sub-graphs dynamically, then how to restructure the graph due to connections change. In general performant, distributed graph technology is still a research problem, and therefore there are not many stable distributed graph solutions.
The most mature of the graph solutions would be the *Titan* supporting Cassandra and HBase storage. Also Google has made some effort to develop a graph engine called *Pregel*. The model applied to traverse the graph is called *Bulk Synchronous Parallel* (BSP). With BSP the compute model is based on moving in steps from each node to all links to the next nodes, and later to the next nodes etc. Actually the same BSP model is used in two other products *Apache Hama* and *Giraph*.
Regarding solutions based on the distributed FS the *GraphX* should also be mentioned. GraphX is an engine for graph algorithms on top of *Spark* as an underline compute engine and the distributed FS as the storage.

Logic Programming In this area we represent queries as logic statements. The *Datalog* representative solution is not really new as it was originally implemented as a subset of *Prolog*. It is actually an alternative to the SQL relational model. The most recent examples are *Datomic* that uses Datalog query model and *Cascalog* that is a *Clojure* dialect [61] on top of *Cascading* which provides one of the most concise statements, even compared to Scala or SQL, e.g. in terms of the *world count* task.

Probabilistic Programming The area of *probabilistic programming* is a good example to a trend to de-emphasing data model and letting the fundamental algorithms to process the data. One of the appliances are the *probabilistic graphical models* for inherently probabilistic systems e.g. weather forecasting or medical diagnostic predictions about diseases based on symptoms. The example of this approach are *Bayesian Networks*. Here the model involves probable causes and symptoms that are likely to take place, leading to outcomes. E.g. M.D. can observe symptoms, and then can use the model and infer the probability that is caused by a disease. This approach is used for modelling some areas where there are no absolute answers, based only on certain outcomes and symptoms and infer the probability of likeliness of precise facts or scenarios. Another example is *Markov Chains* or the *Monte Carlo Markov Chain* used for modelling the sequence of events where the probability of the next event depends on one or more previous

[61] Created by Nathan Marz – the creator of *Lambda architecture*. (Lambda architecture was designed for massive data volumes with the batch- and the stream -processing. It is based on providing accurate views of batch data with batch processing (balancing latency, throughput, and fault-tolerance), while simultaneously providing real-time stream processing for viewing online data. Thus the use of streaming map-reduce latency is being reduced.)

events. The difference is that the Monte Carlo case is based on a simplified assumption that only the current event matters when predicting the next one regardless of what happened in the past. The modern autonomic driving cars can be an example of its appliance when the map and the sensor data are combined with a probability of being real. This is never a precise mapping of real-to-virtual view due to measuring errors, real-world object not on the map etc. Thus we deal with inferring a probabilistic view of the car location and real world surrounding.
The related project originated at Berkeley is called *BlinkDB*. This is a massively parallel, approximate query engine for interactive SQL over large data sets. It enables to sacrifice query accuracy for the response time. Therefore the answers are rather approximate than absolute.

Dataflow Programming Cascading / Scalding (Scala-based equivalent) or Spark API tools encourage a dataflow-style model that is based on pipelines of data streamed through. An example of defining the dataflow that internally handles synchronizing data state. An exemplary implementation of dataflow programming, written in Haskel and JavaScript, is called *Functional Reactive Programming*. It empowers synchronizing mutable states over large datasets. It is especially practical when considering the model for Convergent Replicated Data Types (CRDT) [84, 85] for synchronizing shared mutable state at scale. In such case concurrent updates can actually commute and it is not possible to assure execution of the updates by all replicas in a correct order of arrival timestamp. Thus the issues of operations while concurrent updates commute and all replicas execute all updates in casual order are solved. Currently it is being implemented in Riak and Akka.

2.3.9 Database taxonomy

The dissertation present only a draft of available real scale dedicated and emerging new sources. The scale is, however, enormous and recently one can observe a noticeable increase in evolutionary changes in the area of data storing solutions. To give just a brief overview of how the present situation looks like, a schema was attached to acknowledge the current state of evolution in the data storing solutions – see Figure 2.10. From the economic point of view one should consider

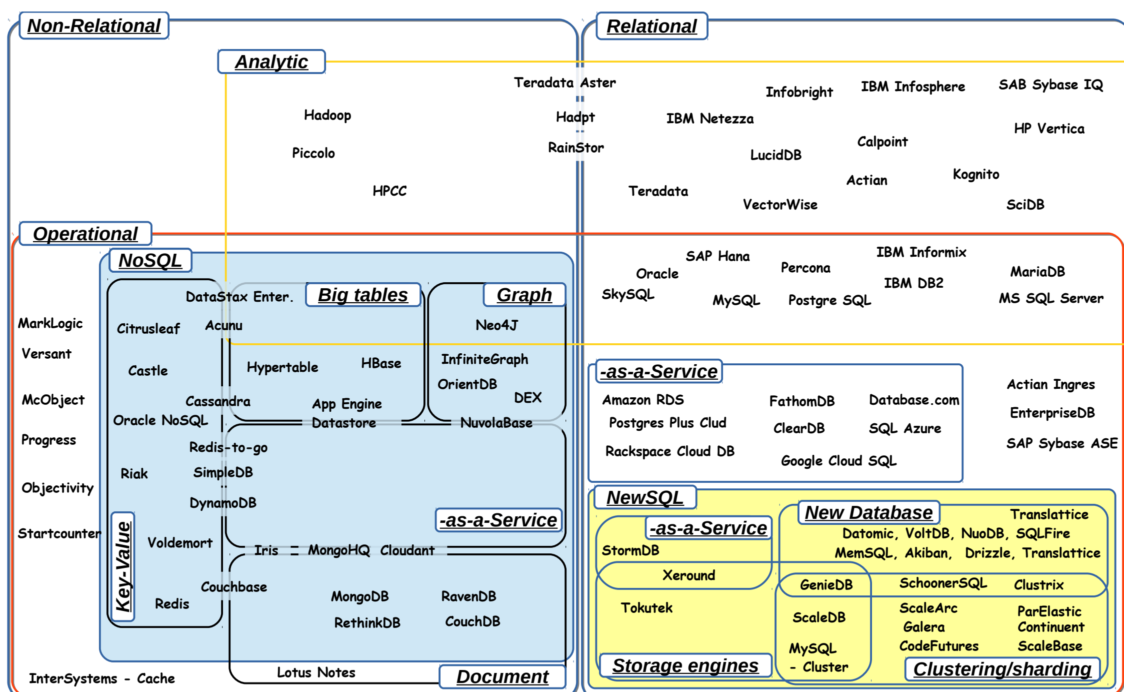


Figure 2.10: Database landscape as of 2015

all aspects of storage, integration, processing and analytics of the data. The total number of essential factors is important and requires some classification. The data solutions dealing with the mentioned aspects must be considered regardles of the data nature itself. The areas that should be considered while deciding about economics should be considered individually for each business case – see Figure 2.11

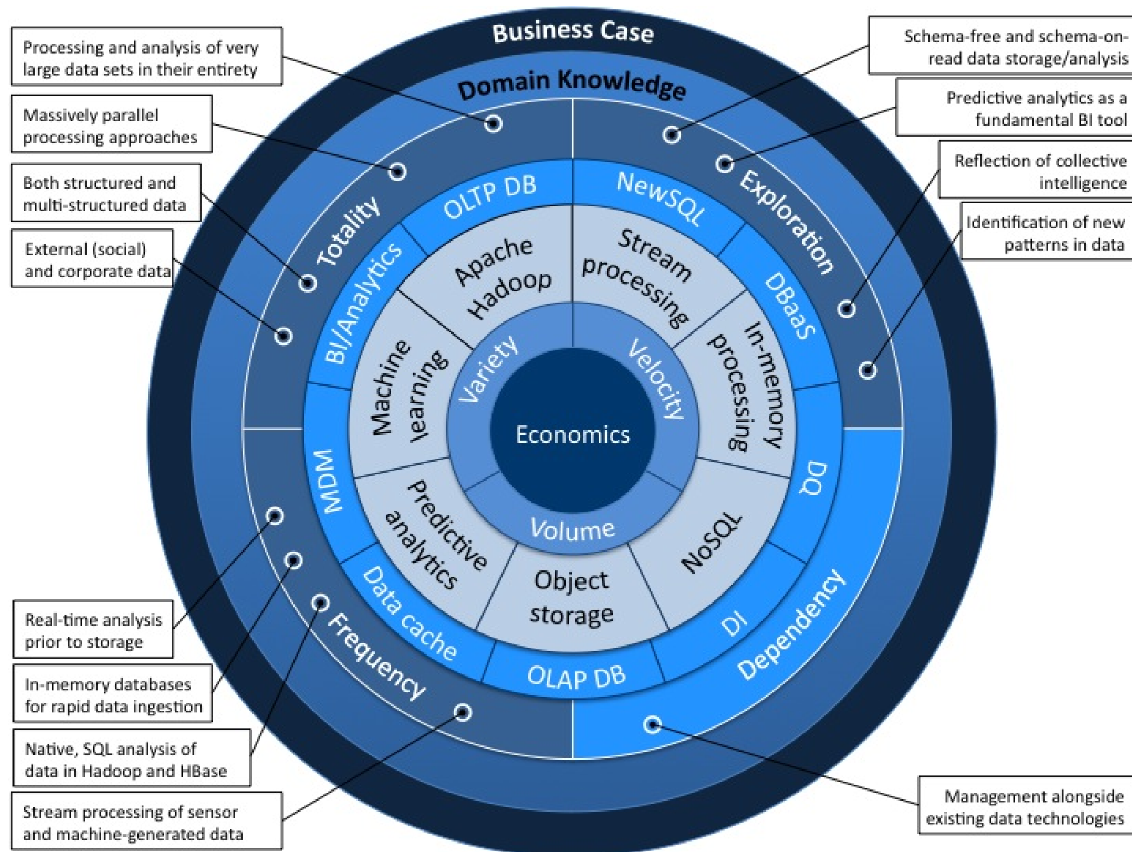


Figure 2.11: BigData vs Economy [86]

2.4 Related Works - Overview of Modern Integrating Solutions

Integrating distributed data and service resources are an ultimate goal of many current technologies, including distributed and federated databases, brokers based on the CORBA standard [87], Sun's RMI, P2P technologies, grid technologies, Web Services [2], Sun's JINI[3], virtual repositories [4], metacomputing federations [5, 6] and perhaps others. The distribution of resources has desirable features such as autonomic maintenance and administration of local data and services, unlimited scalability due to many servers, avoiding global failures, supporting security and privacy, etc. On the other hand, there is a need for global processing of distributed resources that treats them as a centralized repository with resource location and implementation transparency. Distributed resources are often developed independently (with no central management) thus with the high probability they are heterogeneous, that is, incompatible concerning, in particular, local database schemas, naming of resource entities, coding of values and access methods. There are methods to deal with heterogeneity, in particular, federated databases, brokers based on the CORBA standard and virtual repositories. If a global defragmented collection is very large (millions or billions of records), it would be practically impossible to process it sequentially record-by-record. Hence some query optimizing mechanisms are required. For instance, some optimization methods such as indexes and query rewriting are described respectively in [88, 89]

and in [90]. To some extent, these methods can also be applied to distributed heterogeneous databases with the fragmentation transparency, but their scope is limited. Distribution, heterogeneity and defragmentation concerning arbitrary model databases imply very challenging optimization problems which practically have not been considered in a holistic way so far in the database literature.

2.4.1 OLTP & OLAP - sets of operations

Overall database activities can be divided into two broad classes. The first, traditional one - is called *Online Transaction Processing* (OLTP). It involves short querying, inserting and updating transactions, simple queries dealing with small portions of data, while still allowing small, frequent and fast queries. The very fast and effective (transactions per second) query processing, maintaining data integrity in multi-access environments is the main emphasis of OLTP. OLTP only considers current and detailed data based on entity model (usually third normal form - 3NF)
The second and more recent one is *Online Analytical Processing* (OLAP) [62] . Its characteristic is rather opposite to OLTP with low volume of long transactions involving complex queries with aggregations that result from rather large volumes of data, rarely or never being updated. Its effectiveness is measured with the response time. The data that it uses is historical, aggregated and stored in multidimensional schemas.
Those are two, quite extreme definitions with broad spectrum in between. This includes moder-

Nr	Property	Operational Database (OLTP)	Data Warehouse (OLAP)
1	Source of the Data	Every day processing	Heterogeneous OLTP databases
2	System Goal	Fundamental business jobs	Supporting decisions, planning, analysis, data mining
3	Data Date	Current data processing	Historical information processing
4	Users	DBAs, database professionals, db clients	Knowledge discovery, analysts, managers, executives
5	Number of Users	Thousands	Hundreds
6	Goal	Snapshot of current business processes	Set up of data with analytical significance
7	Outcome Focus	Data is output oriented	Data is application oriented
8	Data Nature	highly detailed and primitive data	Consolidated and summarized data
9	Data Structure	Flat relational data schema	Multidimensional views – hypercubes
10	Database Design	Highly normalized, multiple tables	Typically de-normalized with fewer tables In star/snowflake/constellation schema
11	Inserts/Updates	Run by user; fast and short	Run periodically; long batch jobs refreshing data
12	Queries	Return few records; simple and standardized	Return millions of records; complex with aggregations
13	Speed	Very fast	Depends on complexity of input data; might take even a few hours
14	Volume Size	Relatively small due to only current data storage	Integrates multiple OLTP with intense aggregation and historical data thus large and optimization (e.g. index) dependent use
15	Overall Characteristics	High performance	High flexibility

Figure 2.12: OLTP vs. OLAP collation.

ate amounts of data, update queries or the transaction complexity. However, the problem was that right from the beginning the databases were designed to meet the OLTP requirements. Over the years additional techniques had to be developed though, to assure the OLAP support. Thus, presently DBMSs can be tuned up to be compatible with the OLAP requirements.

[62] The term was coined by E.F.Codd himself in [91].

DEFINITION 2.12: *Online Analytical Process (OLAP)*

OLAP - set of operations [a] /approach that enables processing of multi-dimensional analytical **queries/operations** on OLAP hypercubes representing the data prepared in data warehouse or other data source. Part of *Buisness Intelligence*.

[a]by *Gartner*[82]
[a]pivoting, slicing, dicing, drilling

This kind of software model is especially popular with the enterprise class solutions. It is due to large sizes of such systems and numerous operational data sources (that often occur to be numerous OLTP databases), that are expected to provide integral analysis and reports.

DEFINITION 2.13: *Business Intelligence* (BI)

Business Intelligence refers to collecting business data to find information primarily through asking questions, reporting, and online analytical processes. It is a process of collecting data, transforming it to information and obtaining knowledge out of this information.

OLAP is a part of *Buisness Intelligence* (BI) which is gaining new insights about business or markets by dint of technology, systems and good practices for analysing crucial business data (like the BigData, text or network [92]). This gives obvious gains in the form of improving efficiency, workflow and finally the product.

Here the author is obliged to explain how one can distinguish between the BigData and the BI regarding data and its usage.

INFOBOX 5: BIGDATA VS BUSINESS INTELLIGENCE

The BigData use *inductive statistics* [a] to infer rules [b] from large data sets with low schema information density to unveil relationships and dependencies. It can also try to predict behaviours and outcomes. On the other hand, the BI uses high information density data, measuring and detecting trends with the use of *descriptive statistics*. Therefore, the BI aims at summarizing the target sample data rather than extrapolating and learning about the entire data based on the data sample.

[a]Dealing with conclusions, generalizations, predictions, and estimations based on the data from samples.
[b]Often uses also non-linear system identification method for identifying or measuring the mathematical model of a system using the IO measures. It is categorised into four approach models of neural network, Volterra series, block-structured and NARMAX.

2.4.1.1 Data Warehouses - Integration and Analysis

Through proliferating data sources a need of data integration entrenched in modern computer world. What is more, exponential growth of stored data and new applications for databases have enhanced this effect even more. The collected data required a solution for a holistic governance of analytical summaries. A kind of workflow where the data is brought from the operational OLTP sources into single "warehouse", for the OLAP analysis, has emerged and called *data warehousing*. The reason for OLAP emergence was that issuing complex OLAP queries against OLTP generally would result in unacceptably high costs and thus, low performance. What OLAP has also enabled was heterogeneous integration of historic data and provision of special data organization and access methods not provided by operational, RDBMs serving the OLTP. As a result, the data warehouses become implemented separately from operational databases.

DEFINITION 2.14: *Data Warehouse (DW, DWH) / Enterprise Data Warehouse (EDW)*

Data Warehouse - store used to bring together data from different data sources in easy for analysis format and optimized for further OLAP analysis.

Based on the warehousing analysis, in the enterprise scale systems an infrastructure of *Decision Support Systems* (DSS) is also often introduced. In such cases the data collected from multiple databases is stored in specially tuned - for OLAP analysis - data warehouse, which is referred to as the DSS. There can be basically data-, document-, knowledge- or model-driven systems depending on the needs and target area of requested decision making based on data analysis. On the whole what DSS consists of [93, 94], is:

- database (most often numerous with data crucial to decision making)
- the model that determines the context of the decision and the criteria
- the user interface

Basically there are three main applications for data warehouses:

- Information Processing - Querying, statistical analysis, reporting with tables, charts and graphs of data processed within data warehouse.
- Analytical Processing - The analysis made with the use of slice/dice, drill down/up, and pivoting OLAP operations.
- Data mining - Using the data warehouses for knowledge discovery using hidden patterns, associations, building analytical models or performing predictions and classifications.

The following chapters of this dissertation will include a in-depth discussion of a *Cuboid* integration concept that might get confused with OLAP cube. Therefore, for the sake of discrimination between those two terms, in the following paragraphs the author presents a detailed description of what OLAP application and design principles are. The detailed comparison and contrast analysis will be enclosed in the chapter dedicated to Cuboid based architecture.

⋆ ⋆ ⋆

OLAP solutions are mostly based on a type of relational schema named *Star Schema / Dimensional Model* [63] (as proposed by Ralph Kimball) , *snowflake schema* [64] or *fact constellation* which is actually a collection of star schemas that contains multiple *fact* tables sharing many *dimension* tables. Typically it includes one very large table named *fact table* with references to many smaller tables of the schema. This table is a subject of frequent updates [65]. The rest of the star schema tables are called *dimension tables* that in contrast are not that large as the fact table and tend to get updated infrequently. The most common case is that the fact table is based on inserts only originating from the dimension tables – which store more basic or real world data, compared to reporting/logging nature of the fact table. The fact table attributes can be divided into two groups. The first group consisting of foreign keys to the dimension tables is called *dimension attributes*. The rest of the attributes are called *dependent attributes* due to being dependent on the values of the dimension attributes. In the fact table you can also have non-key values that can

[63] In this schema the dimensions are *denormalized*- one table, one dimension. Owing to this, we have fast aggregations and simpler joining queries compared to *normalized* alternatives. On the other, a hand lack of *normalization* results in no enforcement for data integrity.

[64] The normalization of this multidimensional database originates in removing low cardinality attributes by creating new tables [95]. In this form, retrieving results involve increased complexity of query joins comparing to *denormalized* case.

[65] However, one has to be aware that OLAP query session often works only on a snapshot and therefore no updates would be commenced.

be a target of aggregation when analysing based on criteria in the dimensions. Those are called *measures*. The SQL-based, OLAP queries, executed over the OLAP star schema usually tend to aggregate on the dependent attributes. Basically they tend to expand the fact table data with use of foreign key to the dimension tables, using joining, filtering, grouping and aggregating. Indeed, the JOIN operation is always costly regarding the performance especially when considering large data size as a result of being consolidated from multiple sources. To solve this problem a specially designed query optimization [96], indexing [97–99], compression [100, 101], forecasting [102, 103] or pre-aggregating techniques – e.g. with intensive utilization of materialized views [66] – have been developed over last fifteen years.

The second approach, apart from the dimensional model, is the *Third Normal Form* (3NF) approach [104]. It declares that the data warehouse should be modelled with the E-R normalized model. Comparing the two models – the dimensional approach seems to model in a more intuitive way the business domain for the users, and moreover, the lack of complexity results in better performance. However, preserving the integrity of facts and dimensions while loading new data from the multiple data sources is complicated. What is more, in the case when business domain requires fundamental changes in functioning, applying new rules will become a serious issue.

The normalized model is based on normalization rules described already in 2.3.1. The subject areas representation is divided into tables grouped by categories and stored in the relational database. This makes adding new data easy and simple. On the other hand, the complicated schema of tables in large business use cases cause multiple heavy join operations requests.

Both models can be represented in the E-R diagram and both involve relational tables joins. The difference is the degree of normalization. However, some research [105] has been made to show that even with the same fields used in both models the normalized ones provide far more information than their dimensional equivalents, however, at the cost of bad usability [67].

Integrating Heterogeneous Databases with Data Warehouses The data stored within the data warehouse originates from multiple and various databases. In this manner those data sources can be called heterogeneous. To gather such information the DWs have two approaches: query-based and update-based. The first one represents the traditional and already mentioned wrapper/integrator (a.k.a mediators) method on top of integrated databases. The query method is as fallows:

- When a query is committed on client's side, it is mapped to an adequate form, using the metadata dictionary, and transmitted to an adequate set of database sources
- At the database site each query is processed by the local query engine
- Each site result set is sent to the data warehouse and integrated into the global result set

However, such an approach, apart from being straight forward, results in serious disadvantages:

- Enabling heterogeneous querying requires complex integration and filtering processes
- All the intermediate steps require time and makes it inefficient
- The higher the frequency of querying, the more expensive it becomes
- Aggregation queries also become very expensive due to such an approach

A successor and modern way of integration present in today's data warehouses are based on the update-based architecture. This model assumes that all data from the data sources is going to be stored at the warehouse site and pre-fetched from the integrated data sources. Thus direct data queries and analysis are possible. This brought some advantages:

[66] In the cases of many repetitive requests/queries and a few updates for the data.

[67] The volume of information was measured in terms of entropy as defined in 2.1. The usability data transformation measure was defined in terms of network theory of *"Small Worlds"*.

- The most important is that the network overhead was eliminated due to local store and querying, thus allowing high performance
- Data retrieval, manipulation, integration, restructuring and pre-analytics are performed prior to the client's request, thus the answer is ready before the question is asked
- No query evaluation at local sources is expected as all the workload is kept at the data warehouse side

Metadata - the DNA of the Data Warehouse Integration Process One of the most important concepts in terms of data integration and manipulation is the *metadata* [68]. In short, metadata is the information in the form of data that is used to represent the target data. As of the data warehouse its crucial goal [106] is to house standardized, structured, consistent, integral data collected from the heterogeneous data systems across distributed resources. It requires an efficient, structural and common approach to provide a grid wide perspective ready to meet reporting and analytical requirements. Thus the metadata is considered important and even referred to as the *"DNA of the data warehouses"* [107]. The applications of metadata are also extensive on the ground of data warehouses and can (as mentioned in [107, pp.116-117]) be divided into: *technical metadata*, *business metadata* and *process metadata* categories. The Business Metadata and Process Metadata are irrelevant from the point of view of integration. Business Metadata describe the data warehouse local data characteristics of: where it comes from, what their purpose is in the warehouse and how it is related to the remaining local data. Whereas the Process Metadata contains logging information generated during the ETL processes, which is useful while analysing query execution process issues thus, it becomes a business metadata for the fact and dimensions tables. From the integration point of view, the most relevant is the technical metadata. In general, it describes the inner store of data warehouse like dimensions, measures, and data mining models. However, it is also responsible for storing data structures of the integrated relational data source, like its: schema (including tables, attributes and their types), indexes, partitions etc. Metadata is about controlling the quality of data entering the data stream by assuring unified view metadata policies compliance. This was considered so important that even a standard was developed by OMG's *Common Warehouse Metadata Interchange* (CWMI) [69] (See [108, 109]) that proposes a way to metadata interchange in the distributed heterogeneous environments including metadata repositories for data warehouses.

This kind of metadata is stored in a metadata repository/manager that is a part of data warehouse build system, and is responsible for governing the extraction, transformation and loading (ETL) target data into the data warehouse. The metadata can be manually obtained or automatically generated from the scanning process through the SQL catalogues and other metadata sources. The responsibility of the metadata repository is to manage metadata considering its multiple aspects, of which the most interesting – in the scope of this dissertation – is integration. Mapping from the operational environments to the data warehouses requires extraction of source contents –database data – and its partitioning, transformation rules, data refresh and also rules of purging.

Despite the fact that metadata importance in data integration can not be overestimated, there are no industry-wide accepted standards and data management vendors considering provision of solutions for only a narrow spectrum of appliances. This results in unaccepted methods for interchanging metadata globally and between heterogeneous sites.

[68] As the metadata plays significant role in the solution proposed by the author in this dissertation its current data warehouse applications will be elaborated here in more detail.

[69] It worth of mentioning that it correlated in time with the *Sarbanes–Oxley Act* that was acted as a reaction to major corporate scandals including Enron and Worldcom. It forced a set of new or enhanced standards for all U.S. public company boards, management and public accounting firms.

Populating Source Data – Extract, Transform and Load (ETL) The first basic step in functioning of the data warehouse is the ETL phase. It is governed by the *Load Manager*. The first step is to *Extract* the data from the heterogeneous data sources and operational databases. The connection is being done with client SQL queries using gateways like the Open Database Connection(ODBC) or Java Database Connection (JDBC). Then the result, in the unified format, is fast loaded into temporary data source. During this process, data is validated to meet the expected domain values. If the validation fails the data is rejected and analysed towards the source system to repair the incorrect records. It is also possible to modify data-validation rules to fit specific characteristics not considered from the beginning.
The second – *Transform* – phase applies a series of rules and functions against the results of the first – extract phase. This includes covering character sets incompatibility between the systems, cleaning the extracted data from the unwanted parts like removing unwanted columns, encoding the free-form values like the gender format, joining data from multiple sources, splitting column into multiple columns while covering separator based lists etc.
The final stage is the *Load* phase when the transformed data is ready to be loaded into the target data warehouse database. This addition can be done on: override, append or time interval bases.

Two models of data storage – dimensional and normalized In general we have a simple categorisation of OLAP based on the data storing model. *Multi-dimensional Online Analytical Processing* (MOLAP) uses a multidimensional array storage with pre-computed information in the form of a *data cube*. This fact of storing all pre-combined possible data requests make the solution very fast responses. However, the step of pre-computation is often very resource-consuming and can involve data redundancy. MOLAP also provides indexing and storage compression [110] thus, requires less storage space. In contrast the *Relational OLAP* (ROLAP) uses the relational databases as a data source. Based on the relational paradigms no pre-calculation is needed here and arbitrary query can be committed and will become SQL query. ROLAP is highly database-dependent therefore its scalability and efficiency are based on the underlying solution and the way it was implemented into ROLAP.

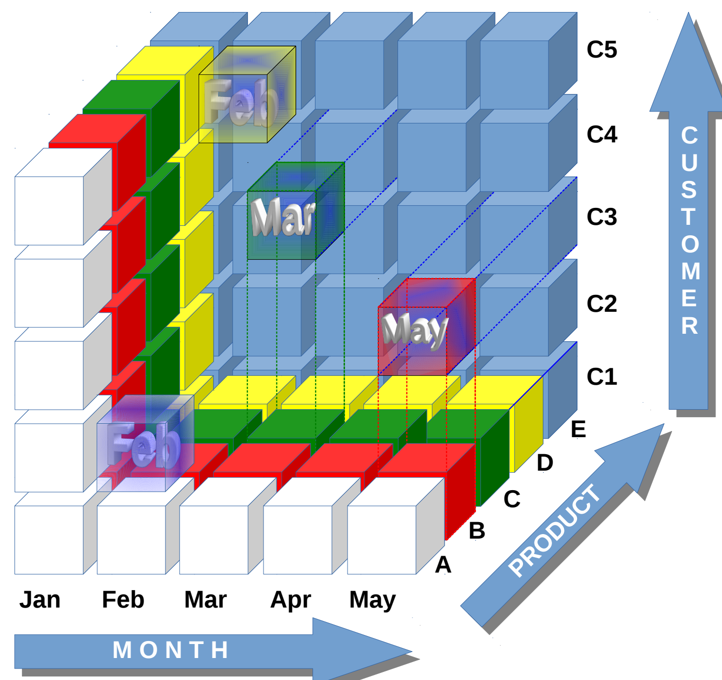


Figure 2.13: OLAP cube example.

OLAP applications have introduced a new way of looking at the data - called data *cube*

a.k.a. *MultiDimentional OLAP* (MOLAP). The nature of the schemas stored at the OLAP sources in the form of *dimension* tables in a straightforward way can be associated with the physical dimensions of space. Thus it is easy to create a 3D representation in the form of cube, with the axis represented by *dimensions*. Of course, it is possible to correlate more than three dimensions and have an arbitrary number of axes that give us a hypercube. Now the content of such a cube is divided into the form of n-dimensional subcubes that can be considered as cells, whose value is defined by the *dependent* (or *fact*) data. The hypercube is spanned by the dimensions of vector space. The intersections of hypercube contain the *measures* - which is how numeric *facts* categorised by dimensions are called. Moreover, the aggregated data values are situated on the sides, edges and corners of the cube. The requested data is obtained by performing aggregations or projections along the dimensions. This is an important reason why OLAP operations outperform the ROLTP solutions [111] greatly for complex analytical queries. All possible aggregations provided by the OLAP can be determined by the number of all possible combinations of dimension granularities. Every single change on granularity of a fact table specific dimension results with a set of data that can be aggregated up, along this dimension. The chosen [70] materialized views, as the form of optimization, expressing the pre-calculated aggregations, can be determined by the number of pre-defined aggregations or time required to update them from the changes taken at the base data store. The materialized views goal is to shorten the time needed for OLAP query result data acquisition.

OLAP Operations The multidimmensional approach enables an extended view of the data but is also a good target for specialized analytical operations:

- *Slice* - picking one value for one dimension and reducing the number of dimensions to gather information only for remaining dimensions related to the given value
- *Dice* - pick a subcube by choosing specific values for different dimensions
- *Drill-Down* - is to move along the range of dimension hierarchy to the most detailed one (down).
- *Roll-Up* - reverse operation of Drill-Down; involves data aggregation along dimension – e.g. computing totals along hierarchy like stating `Locations` dimension as `Countries` instead of detailed `Cities` that is lower in the location hierarchy
- *Pivot* - enables cube rotations in order to provide an alternative presentation of data

Data Warehouses Issues with Modern Data The problem with the data warehouses, however, is extremely fast data volume growth that doubled its magnitude in the last few years. It becomes more and more difficult to store cumulative amounts of such big data in one data warehouse. Moreover, some dedicated sources now include not only textual data, that is easy to manipulate and search, but provide dedicated mechanisms for multimedia BLOBs. Some new, modern solutions, however, proved to deal quite well with these issues (See 2.4.3).

2.4.2 Metamodels - Metadata

The metadata term was first coined in 1968 [113] and its basic meaning was: a data, providing information, about the additional aspects of the target data. Since then it has been widely and continuously adopted as an important part of multiple data oriented software, especially along the distributed and web-based solutions. Metadata is composed of multiple aspects that influence on its categorisation due to different functions they support. The broad spectrum of metadata appliance urged standardisation definitions to be stated. This process originated from the metadata taxonomy, based on its structure and functions.

[70] The problem of choosing the optimal, pre-calculated aggregation set for solving given problems and potential OLAP queries is stated to be *NP-Complete* [112].

2.4.2.1 Metadata taxonomy

The main categorisations of metadata are based on three alike approaches. In the first one [114], there is a distinction between raw architectural description of database tables, columns, keys etc. – called the *structural metadata*, and the *guiding metadata* defined by a set of keywords in natural language that helps the user find requested data. The second (already mentioned [107] in Section 2.4.1.1) approach has renamed this to *technical metadata* focused around internal mechanisms and *business metadata* - dealing with the business external processes. Additionally, the process workflow data has also been categorised as *process metadata*. Finally, *National Information Standards Organization* (NISO) introduced its approach in [115] that defines the following categories of metdata:

- *Descriptive* - enables discovery, identification and retrieval location of the information requested from distributed data sources by the clients
- *Structural* - describes the technical processes of finding according to the schema the data is organized – e.g. sorted
- *Administrative* - preserves the data source management information about the file type and its creation particularities - also known as the meta-metadata
  - *Rights Management* - defines the intellectual property rights
  - *Preservation* - preserving integrity and safety information – e.g. checksum calculations

2.4.2.2 Designing Metadata

The complete metadata statements assembly process requires predefined vocabularies based on standards and metadata modelling approach. This way a complete metadata scheme can be stated. This is especially important while considering the data model for stiff schemas of database designs. Syntax of each schema statement, conforms to the rules created by the structure of the meta-content fields. This enables ease of schema representation with an arbitrary markup language like XML, HTML or RDF [71] . The schemas can also be classified by the adopted schema model. This includes hierarchical nesting of elements in the parent-child relation model (e.g. [116]). The simplest form of metadata schemas are those dimensional, linear schemas with totally discrete elements like [117]. Moreover, planar schemas with the elements based on two orthogonal dimensions and specialised applications (like GIS) with more than two dimensions are also present. The more complex structure of the target data is, the more complex schema structure is required. Data with a deeply structured nature in those cases requires metadata schemas which are refereed to as the high *granularity* schemas. This enables encapsulating more detailed information within such a schema but with relatively higher costs of creation process.

Metadata Modelling Prior to designing the dedicated schema type, all aspects of predefined class of problems that are about to be represented needs to be carefully considered. This modelling of metadata schema involves defining and analysis of rules, constraints and relations by adopting concept generalization, associations, multiplicities and aggregations. The modelling should also consider existing tools and standards that have already been proposed and discussed. Such standardized approach would probably be more general and agile in terms of general application, however, it must be carefully investigated against particular use case requirements.

Metadata standardisation There have been numerous standards enabling the best possible description of a resource type for the dedicated need. The appliance spectrum is very wide from:

[71] For instance in case of Dublin Core Metadata Initiative syntax specifications

social science, audiovisual content, archiving, arts, biology, book industry, data warehousing, ecology, education, geographic data systems, up to government public sector organizations etc. (See Figure B.1 for metadata standards). However, one core standard that would be widely accepted and applied has never emerged.

2.4.3 Distributed File Systems - Embracing Scaling Up in Size

As already mentioned in 2.2.2 and 2.4.1.1 data warehouses are very mature solutions with rich SQL analytics. However, the problem is the poor scalability and great costs per terabyte. In class of hundreds-of-terabytes-a-day cases, despite the already mentioned advantages, data warehouses seem insufficient. Therefore, an adequate and cheaper solution would be required. An answer to the web scale data analytics, with petabytes of data, might be a distributed file system solution that seems to supersede data warehouses well enough.

A straightforward choice could be to replace data warehouses with one of the NoSQL solutions like Cassandra or Riak that scales up really well. However, despite that fact, most of the data warehouses users are SQL experts that are not programmers and thus, have to be able to work with SQL. Another barrier would be the SQL-base infrastructure, that is already built up and storing current data. It should be noted that NoSQL solutions are also stores that are only suitable for specific purposes that favour one preferred data model. Therefore, the NoSQL replacement solution for data warehouses, despite being theoretically possible, raises new serious issues that makes it most of the time ineffective. That is where solutions like Hadoop including the MapReduce framework and HDFS or Dremel with the database or GFS based storage might came in.

2.4.3.1 *MapReduce* - Functional Approach to Distributed Data Processing

In 2004 a new programming model for processing parallelizable issues across very big datasets using a large number of computers (nodes), collectively referred to as a *cluster* or a *grid*, was developed and thus providing new abstraction. The model aim was to hide complex details of parallelization of distributed computations, target data distribution and failure handling while dealing with very large and unstructured datasets [118]. The inspiration for this was based on the functional programming paradigm [118] including namely the *map* and the *reduce* primitives. Also the general principles of the *map* and *reduce* model have been known since 1995 in the form of *Message Passing Interface* with its *scatter* and *gather* operations [119]. In contrast, the MapReduce implementation [118] was tailored towards the cluster-based computing environment. Nevertheless the map/reduce functions are not the key feature of the framework. Its most important feature is that it provides fault-tolerance for a variety of applications due to the execution engine optimizations. In general, what is being provided by the system is replication based fault-tolerance, scalability and the algorithm of processing the data with the functions. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

MapReduce - Algorithm Overview Although the details of Google MapReduce implementations are not disclosed an open Hadoop implementation of MapReduce is freely available. It can be considered as a distributed sort-merge engine. The first principle of MapReduce framework is that there is no data model and the input/output data is stored in files (unstructured data) or in the database (structured date). The algorithm expects the user to provide some functions via implementations of appropriate interfaces and/or abstract-classes. That would be obviously *Map()* and *Reduce()* functions. Additionally, three more functions are required. Firstly, a *Reader()* function that reads input data from files and provides them as records. A *Writer()* function that will form the output records and put them into files of distributed file system (DFS). There is also an optional *Combine()* function.

The map function divides each input problem (record) into subproblems in the form of key-value pairs – i.e. the result of map function is a zero or set of value records associated with each key. Now we can process separately every record associated with each key.

$$Map(key1, value1) \longrightarrow list(key2, value2)$$

As of the second – Reduce() – function it works on the subproblems and combines the results. Its output is zero or more records.

$$Reduce(key2, list(value2)) \longrightarrow list(value2)$$

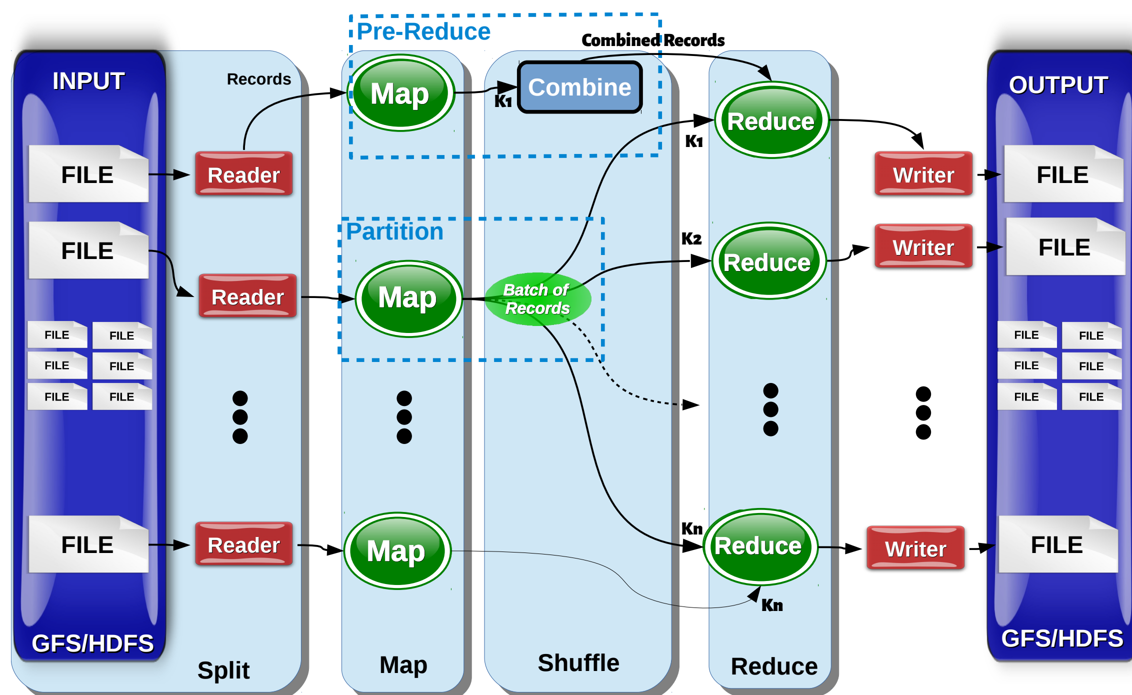


Figure 2.14: MapReduce concept diagram

Let us focus on details of the MapReduce algorithm as explained in Figure 2.14 [72]:

1. *Split- Input reader* - Firstly, algorithm identifies a basic unit of input data that keeps repeating and forms an input record out of it. This includes assigning each map processing unit a key value and all of the input values associated with it. So the stage splits the input files into a number of mapper pieces and starts multiple copies of the program on a cluster machines with one *Master* copy and remaining *Worker* copies.

2. *Run Map() function* - Each worker parses input split file into key/value pairs. The user defined mapping function on each mapper copy, runs once per one key/value pair. Its input is a set of `(input key, value)` pairs and the output is a bag of zero or more `(intermediate key, value)` pairs. The system applies the map function in parallel to all (input key, value) pairs in the input file. The output pairs are buffered in memory.

3. *Shuffle* - The output `(intermediate key, value)` of the *Map()* function is redirected to the adequate reduce processors. It is done as depicted in Figure 2.15. Periodically buffers from the memory are written to the disk and partitioned by the *Partition()* function. But

[72] In *Big Data* the most popular exemplary problem that is solved with MapReduce is *Word Count* which is like a *Hello World* in the case of programming language examples.

before that, an optional pre-aggregation of the data, locally consolidating *Combine()* function – connected with the mapper – is called on the mapper output pairs. As an optional function it is responsible for a pre-reduce phase [73] that aims at taking a set of records for a given key and producing a combined version of such records, and hence it can reduce the data sent to the reducer. This makes things more efficient and causes sending less data over the network. Next the *Partition()* function defines the partition for the records emitted from the map – the partition determines which reducer will process the record. After the `Partition(key)` determines the partition, records are sorted for each partition by the record key with `SortByPartition()` function. Finally, so called spill file is created gathering all the output records. Its content is ordered by partition, and with map output key within each partition. As a result, this information about partitioning stored at the disk is being forwarded by the master to appropriate reduce worker.

4. *Reducer employed* - *Master* worker notifies the reducer worker about the location of the data. The reduce worker reads the remote data from the local map workers disks. Then the reduce worker sorts all data by the *intermediate keys* and groups them.

5. *Reduce() function* - The *Reduce()* function processes for each unique *intermediate key* its values. The resulted output is appended to the output file for each reduce partition.

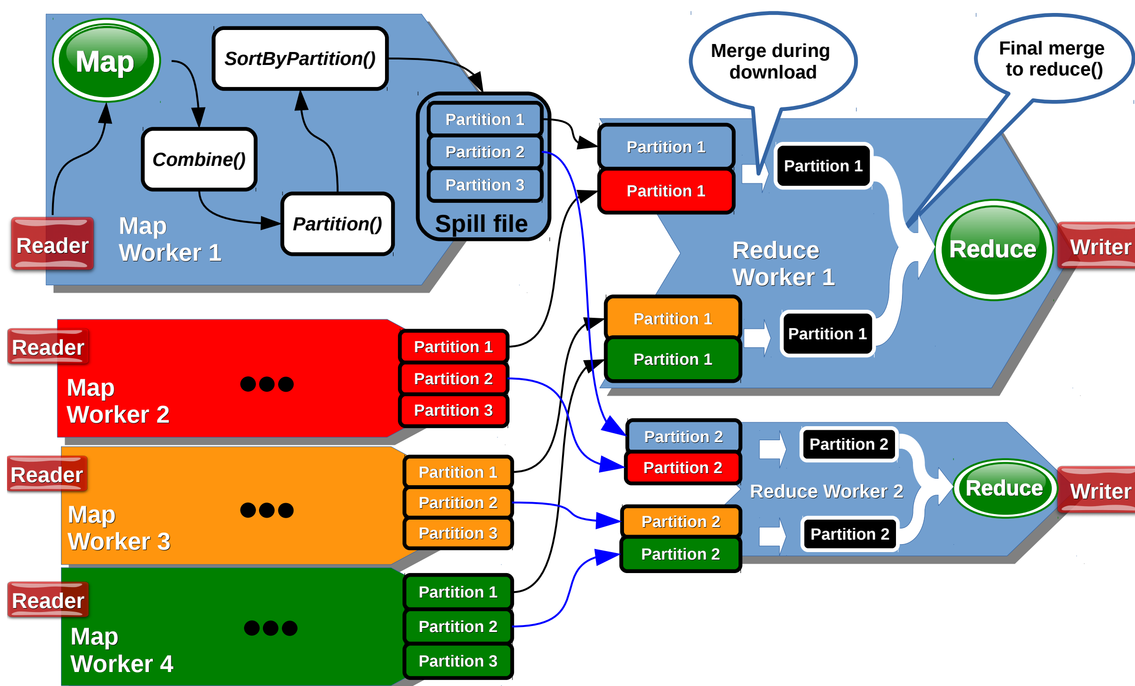


Figure 2.15: MapReduce Shuffle mechanism.

MapReduce - performance There are no guarantees that MapReduce assures speed. Its main advantage is the opportunity to decompose large amounts of data [74] of similar processes into a parallel and distributed manner and provide *transparent fault tolerance*. Moreover, careful consideration of trade-offs between the cost of computation and communication is obligatory [120]. The efficiency of the MapReduce – due to its *distributed-by-design* nature - is based on the cost of the network communication. Communication cost often exceeds that of actual

[73] The Combine() function typically implements the same algorithm as the *Reduce()* functions. The only difference is that its output is not the final output but it becomes a part of a *Reduce()* function input.

[74] Where the data does not fit into memory of a single machine or a small cluster.

computation. Therefore reducing the network traffic where possible is a key to achieve good overall system performance. Right from the first steps of the algorithm some optimization on this field can be commenced. This includes running the data split and the *Map()* function logic at the same node where the data is stored. This is possible while considering Cassandra or HDFS as a store but not e.g. the Amazon S3. So execution of the mapper logic at the storage node is the first optimization consideration that can influence the general performance.
Collocating *Map()* and *Reduce()* at the same node would be effortless because the *reducer* is picked by the key. The only way to optimize the networking overhead is to provide optional *Combine()* function whose goal is to consolidate the amount of local data before sending it to the *reducer*. The most fragile part of the algorithm is writing of the output to the output store. In the case of HDFS, as an output store – due to its replication safety features- it can be a very expensive task. On the other hand, the NoSQL solutions like the Cassandra, enable configurable latency however, at the cost of consistency trade-off. Thus the overall performance of this algorithm step depends mainly on the amounts of data resulting from the *reduce* phase and fortunately this resulting volume is mostly, relatively reasonable.

It should be mentioned that MapReduce in single-threaded implementations will not provide any performance gains compared to non-MapReduce implementations. This can be noted for example, in the case of MongoDB implementation where the price for using MapReduce is speed [121], due to slow grouping. MapReduce is not suitable for real-time either. In exemplary case of MongoDB implementation, the real-time querying is possible only against MapReduce background job resulting collections. Main MapReduce gain could be achieved by running on parallel, however, in the case of MongoDB, it runs on a single server, while parallelizing on shards. Thus, although MongoDB's MapReduce can be executed in parallel at each shard, there are major drawbacks like the *JavaScript* language *SpiderMonkey* implementation that is used by MongoDB, which is not thread safe and thus, again only one MapReduce program can be run at a time. Actually MongoDB built-in MapReduce implementation is several times slower than MapReduce provided by e.g. Hadoop using HDFS as a back-end due to design differences [75] [122].

MapReduce - applications As for the MapReduce spectrum of application scenarios, this is very encouraging. Many existing algorithms from various areas can be morphed into the MapReduce model. It turns out that MapReduce is especially useful while considering parallel data processing design methodology.
MapReduce can be applied to multiple classes of problems. One of them are the *Map-Only* issues that include problems such as *Distributed GREP*, *Document Format Conversion*, *ETL*, *Input Data Sampling* etc. Such problems have no need for consolidate data or aggregating individual results after the map phase, thus no reducer required. This means that the *Map()* results becomes the final results. The other cases are:

- Generating the inverted indexes that involve parsing different documents to build a word search index. It was used to completely change the way Google has processed the WWW index.

- Simple statistics generation of *count, min ,max, avg* etc.

- SQL model can be applied to extract data. E.g. While the *Map()* function can implement the projection (SELECT) and filtering (WHERE), the *Reduce()* phase can be used to implement the aggregation functions(Min,Max, Avg, etc.), grouping (GROUP BY/HAVING).

[75] Like the fact that Hadoop DFS is optimized for sequential reads and writes of data in relatively large chunks whereas MongoDB is optimized for random and parallel access, i.e. queries to the data. What is more, MongoDB turns out to provide unsatisfactory performance regarding parallel writes due to the global write lock.

- Data joins (JOIN) including the reducer-side join, map-side partition join, map-side partition merge join etc. ([123, 124])
- etc.

In general, MapReduce has no data model and data is stored in files, regardless of the data schema. The user is obliged to provide the main algorithm functions. The system provides the system algorithm integration, fault-tolerance and scalability. However, it is obvious that the idea requires low-level programming and notion of database schemas and a declarative querying engine is missing. The solution was later proposed in the form of *Hive* and *Pig* languages situated on the top of MapReduce implementations like Hadoop (See 2.4.3.3).

The MapReduce was investigated against the RDBMS and tested for several specific problems, and apart from the assumption that it has never intended to be treated as a database, it was proved [125] that RDBMSs still can offer desirable features while compared to MapReduce, especially in enterprises. The MapReduce, on the other hand, proves to be easy to implement and adequate for simple or specific processing tasks.

2.4.3.2 Dremel - Interactive Analysis of Web-Scale Datasets

At this point it is obligatory to mention a closed project from Google Inc. named *Dremel*. The project has been in production since 2006 for the Google internal usage. It was designed to support real-time analysis of large datasets over commodity based machines of shared clusters. Dremel uses the distributed *Google File System* (GFS) to execute many queries in just a fraction of the execution time compared to a sequence of jobs required by the MapReduce-based architecture. However, the Dremel usage is not replacement or substitute but rather to complement and reinforce the MapReduce-based computing [126]. Google *BigQuery* is a Dremel based platform for very large data sets interactive analysis stating: tree based query processing, SQL-like semantics and column-oriented storage.
Inspired by Dremel, an Apache TLP *Drill* project started to provide the same functionality within the open source environment. Drill has provided the schema-free ANSI SQL query engine with real-time interactions. As such, Drill became useful Hadoop based rapid application development with BI analytics.

2.4.3.3 *Hadoop* - The Petabyte *"Elephant"* of Distributed Warehouseing

Hadoop is a software MapReduce-based framework that enables integration and distributed processing of large data sets across clusters of computers, using simple programming models. Its goal is to scale up to thousands of servers that are expected to be error prone and thus, Hadoop detects and handles failures at the application layer [76]. Hadoop contains a distributed, scalable, and portable *Hadoop Distributed File System* [77] (HDFS). Hadoop goal is to complement current data center capabilities with large structured and unstructured data set analytics. Regardless of the operating system, Hadoop divides large data sets into smaller pieces that can later be simultaneously processed at multiple nodes stored on regular, cheap servers.

The first and one of the biggest users of Hadoop is Facebook that migrated a few years ago, and moved several hundreds petabyts of data, from data warehouses to Hadoop. Apart from Facebook or Yahoo!, however, Hadoop has been adopted in more than a half of companies [78] from the *Fortune 50* list [127]. The only disadvantage of rapidly developing Hadoop, is the fact that it is still not as mature as data warehouse. However, apart from that, it has the order of magnitude lower costs of scaling the data size (as of [37]) and – unlike NoSQL – SQL support.

[76] Failure handling can be done by repeating the failed task at another node.

[77] The file system uses TCP/IP sockets for communication. Clients use the remote procedure call (RPC) to communicate between each other.

[78] To name just a couple of the most recognisable: Amazon, Facebook, Yahoo!, last.fm, New York Times etc.

On the other hand, Hadoop is not a solution to transaction handling. One transaction started on a distributed system can generate many additional operations that would have to be rapidly executed - which Hadoop can not assure. Therefore, low latency systems are also a no-go for Hadoop. However, Hadoop can replace sometimes the ETL tools. For instance, let us consider an enterprise website with trade transactions being transferred with ETL to the data warehouse every night. The size of the data, after the ETL transformation, decreases by about five times compared to the actual transaction data volume. So the ETL process has made some of the data irreversibly lost and thus impossible to be additionally analysed that was not considered at the time of launching the ETL task. The opportunity to store such transactional data in HDFS, replacing the ETL with *MapRaduce* analysis and finally sending the results to data warehouse, shortens the analysis time and preserves all of the additional data for further, future processing if requested.

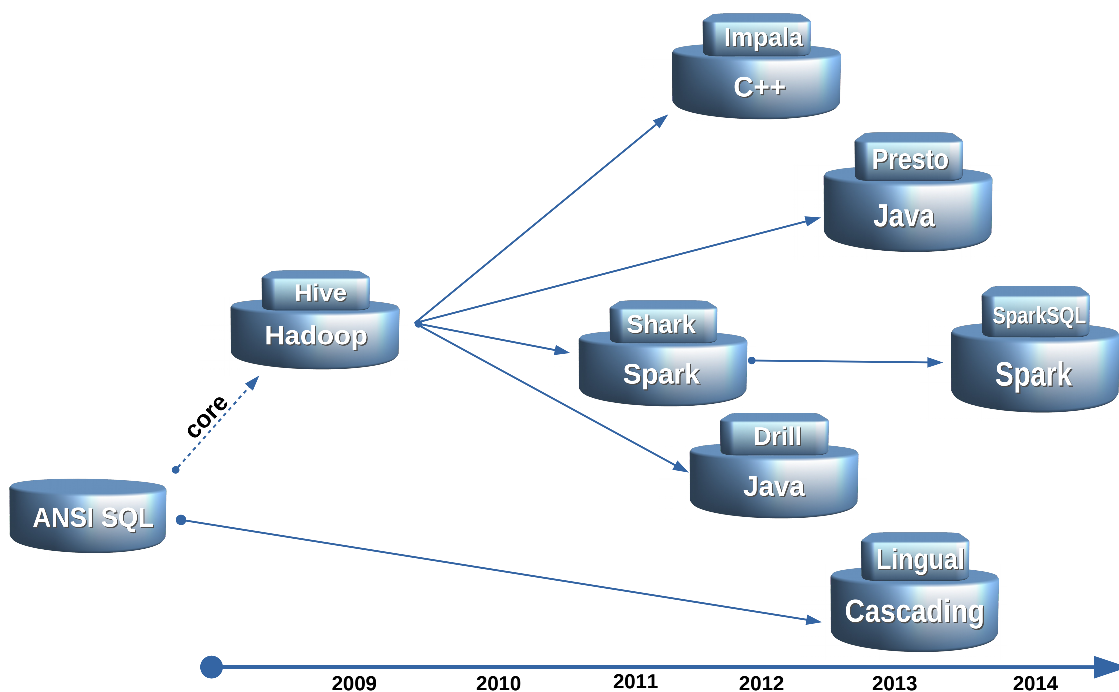


Figure 2.16: The SQL-like query engines emergence.

Considering the fact of the SQL support on Hadoo,p makes it the best solution across industry in terms of scale cost per terabyte. The re-emergence of SQL in terms of Hadoop solution (see 2.16) raised a strong alternative for the data warehouses. The response to this is *Apache Hive* – an SQL like interface to the data stored in a Hadoop cluster. It uses MapReduce with rather only core implementation of ANSI SQL-92 [79] . However, it keeps developing and the attempts are made to comply the full ANSI standard. The missing schemas and declarative query language were introduced in form of the *Hive* and *Pig* that were able to use Hadoop and provide the missing functionalities. Hive has provided the schemas and a SQL-like query language called HiveQL. Hive allowed mapping the HDFS files into Hive tables or use the HBase [80] tables and then query such data. Thus, no MapReduce jobs writing was required. The HiveQL under the hood is converted into MapReduce jobs and then run on the cluster to give the results. On the other hand, Pig [81] , as another example, has introduced a data flow language with more imperative, statement syntax based on relational operators that show some resemblance to

[79] HiveQL supports neither transactions nor materialized views, and only has limited support for subqueries [128].

[80] Released HBase 1.0 on 24 February 2015, after seven-year development attempt.

[81] Originally developed at Yahoo.

the statements of relational algebra. Pig allows you to process enormous amounts of data very easily and quickly by repeatedly transforming it in steps and thus simplifying the writing of MapReduce jobs. Both languages compile into the graph of MapReduce Hadoop jobs, where job is one instance of a MapReduce.
Another system enabling generating similar workflow as Hive and Pig would be the *Dryad* with its own laguage *DryadLINQ*. DryadLINQ compiles into the Dryad system in the same way as Hive/Pig compiles into the workflow of MapReduce jobs.

It must still be remembered that MapReduce implementations like Hadoop are designed for more OLAP-like kind of operations or analytical operations. Thus MapReduce exhibited its poor performance in some areas, mainly due to primarily being designed for batch processing which makes it less powerful for ad-hoc data exploration and machine learning processes.

In 2011, an attempt was made to fix this by using a new compute engine called *Spark* and replacing the MapReduce while keeping the query engine. It builds directly on the Apache Hive code base, so it naturally supports virtually all Hive features like Hive SQL language. This project was called *Shark* and resulted in very good performance [82]; due to Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG) execution engine that Spark is based on. The partial DAG execution (PDE) technique takes advantage of fine-grained data statistics for dynamic queries run-time optimizations. Thus, DAG made it possible to eliminate the MapReduce multi-stage execution model, while retaining the fine-grained fault tolerance properties and still offered even a 100x performance gain [129], comparing to Apache Hive. The extension also involved column-oriented in-memory storage and dynamic mid-query replanning. Moreover, Shark results [129] matched the performance reports for *Massively Parallel Processing* (MPP) database architectures (e.g. Amazon's *Redshift*) that were known [125, 130] to also outperform the Hive speed. This was achieved while providing fault tolerance properties and complex analytics capabilities that MPP lacks. Finally, Shark has the possibility to run based on Hadoop or standalone in the cloud and use any type of Hadoop data source like HDFS, HBase or Cassandra.

The more interestingly HDFS, apart from the Hive based data warehouses MapReduce jobs, can be used for any sort of application that is based on batch and parallel data processing rather than real-time, like the *Mahout* machine learning system or HBase database. The latter one is a *CP* (in terms of CAP-Theorem) type system running on top of HDFS. This is due to HDFS, being a FS, lacks random read and write access. This is where HBase, as a distributed, scalable, big data store, modelled after Google's BigTable, has proven to be a very useful. While using Hadoop as the repository of static data, the HBase plays the role of a datastore that holds the data that is going to be changed over time after ongoing processing. The HBase has become adopted by the largest Hadoop users [131]. Owing to the HDFS based deployment, the HBase as a key-value store enables storing large volume of sparse data in the fault-tolerant way. HBase has become particularly suitable for finding a specific data in large collections of meaningless or empty records. Moreover, features such as compression, in-memory operations and Bloom filters [83] have resulted in the increase of interest in HBase of large companies such as Yahoo, Adobe or Facebook that used it for their messaging service in the last couple of years [131].

Impala - Abandoning Hive The year 2012 brought the Cloudera's initiative called *Impala* (C++ based) [84] . Impala became an enterprise data warehouse system that works well with Hive and HDFS. At the beginning it was considered to *"supplant Hive"* [132], however, this has soon occurred to evolve into dedicated for real-time querying architecture [133].

It was developed to leverage the flexibility and scalability strengths of Hadoop – combining the familiar and low-latency SQL queries support and multi-user performance of a traditional

[82] Executions were up to 100x faster than Hadoop MapReduce in memory, and even 10x faster on disk with use of an advanced DAG execution engine that supports cyclic data flow and in-memory computing.
[83] Space-efficient probabilistic data structure.
[84] Created by former Dremel developers from Google Inc.

analytic database. Impala is a massively parallel processing/MPP-like query engine for the data stored in the Hadoop cluster that bypass the Hadoop MapReduce with its custom query engine running on separate nodes. Impala circumvents the MapReduce and access the data directly through a specialized distributed query engine that is very similar to those in the commercial parallel RDBMSs. The result is the order-of-magnitude faster performance than Apache Hive. The data itself can be stored in HDFS and HBase and queried with HiveQL using the ODBC driver while 3-30x faster [132] [85] than Hive over MapReduce. Impala integration with Hadoop enables the use of the same file [86] and data formats, metadata, role-based authorisation security and resource management frameworks used by MapReduce, Hive, Pig and more. Moreover, Impala enables the use of BI tools to perform analytics via SQL.

Impala, however, is not as mature solution as some concurrent software. Following paragraph proofs a short maturity study on Impala, commenced by contrasting it with Shark. Shark, in contrast to Impala, supports all Hive features [87] due to being built directly on Hive codebase due to Java based implementation. Impala, due to being built with C++ does not support Hive's UDFs, but as well as Shark integrates very well with the BI tools. However, Impala's queries are compiled into *Low Level Virtual Machine* (LLVM) intermediate representations, and thus can be a subject for just-in-time optimizations by the compiler. Shark still lacks query compilation into the JVM bytecode. What Impala does not provide is in-memory data storage. Shark enables performance gains due to query processing on the data preloaded into memory. What is more, Shark uses compressed, column-oriented format for the in-memory data. As for Impala the fault tolerance is also an issue as query restarts are required in the case of node fails which in the case of longer queries is unacceptable. On the other hand, due to its underlying Spark engine Shark can handle mid-query faults. Both solutions have indisputable performance gains over Hive of up to 100x in-memory and 5-10x on the disk faster. Impala appliance is the enterprise OLAP and data warehouses. Whereas, Shark not only supports OLAP, but also more complex Hive usages (like UDFs), processing of unstructured data (e.g. ETL) and – owing to Spark integration – advanced analytics, like machine learning. Thus Shark is about to support not only SQL but also advanced analytics (like statistics, etc.)

The result was that Impala became current fastest, in the Hadoop ecosystem, way to run queries in terms of performance. Some performance results of Impala achieving better concurrent latency than its competitors while providing high query throughput, and with a far smaller CPU footprint can be found in Appendix C. However, due to being developed with C++, it was hard to reuse things already working on Hive (Java based) or generic Hadoop which actually required custom patches for Hadoop jars to work with Impala.

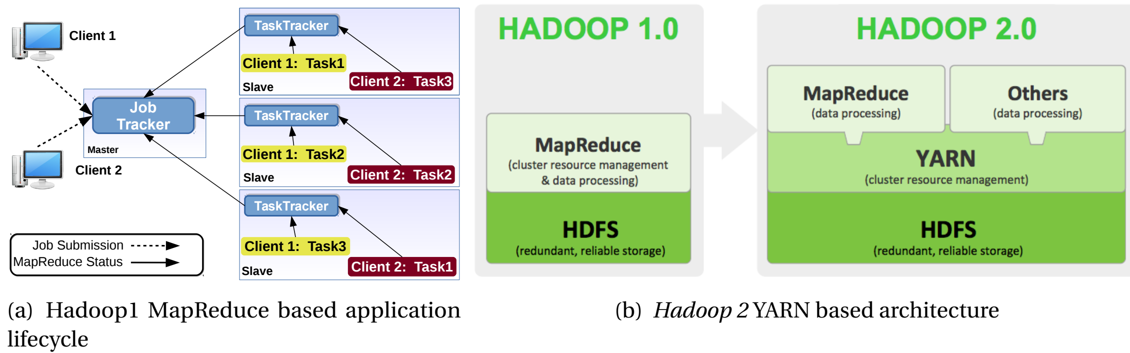
Hadoop 2 The second iteration of the Hadoop framework – in form of *Hadoop 2* released in 2013 had a fundamental importance for further revolution of Hadoop ecosystem. Overall gains considered better performance and stability, however, some major features made this a revolutionary step. The change was conducted in main areas including: new HDFS functionalities, introduction of YARN, reconsidering the MapReduce model and providing heterogeneous storages for HDFS. In general, the *Hadoop 2* improvements has focused around three major areas.

- *New Generalization* (YARN) - A completely new generic platform framework - called *Yet Another Resource Negotiator* (YARN) - as a cluster management technology, has been introduced to run arbitrary distributed applications. YARN facilitates writing arbitrary distributed processing frameworks and applications by providing daemons and APIs. The key functions of YARN were to split MR1's JobTracker major functionalities into

[85] See also Appendix C

[86] Like text, LZO, SequenceFile, Avro, RCFile or Parquet.

[87] This includes HiveQL, Hive data formats, user-defined functions (UDFs) and the use of queries calling external scripts



(a) Hadoop1 MapReduce based application lifecycle

(b) *Hadoop 2* YARN based architecture

Figure 2.17: Hadoop MapReduce generalization with YARN

two daemons. One with the resource management and other with job monitoring and scheduling. The MR2 running the MapReduce framework has now become one of many possible applications that can be run on top of YARN. It is possible because YARN provide a more generic execution model than MR1 and thus, the new, rewritten MR (i.e. MR2) can be run on top of YARN as one of many possible applications. However, YARN is not limited to run only applications that follow the MapReduce model [88] . YARN applications could be Message Passing Interface (MPI), graph processing [89] , simple services and more others that are not related to MapReduce. In general, YARN has decoupled the MR resource management and scheduling from the data processing, thus enabling Hadoop to support a variety of processing approaches. As a result, interactive querying and streaming data applications can now run simultaneously with the MR batch jobs. With YARN, an *application* means a single job in the sense of MR1 jobs or a DAG of jobs.

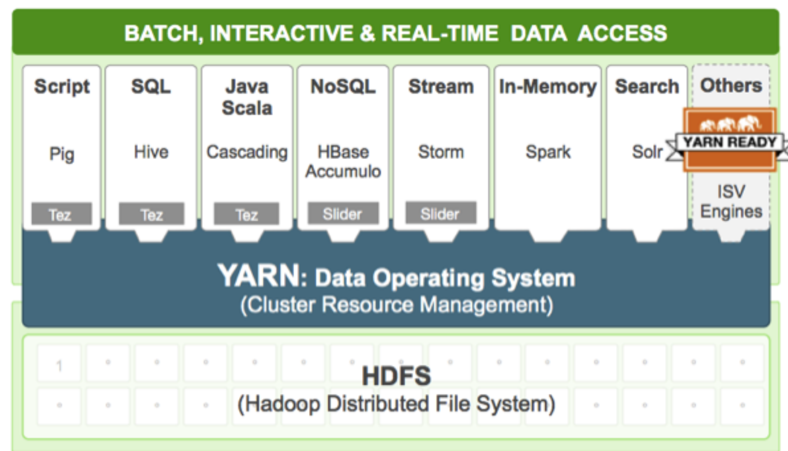


Figure 2.18: YARN-enabled applications running on Hadoop 2.

- *Storage* (HDFS [90]) - Introduction of the HDFSv2 federation has also brought some changes to the HDFS based storage. It involved moving from single to multiple node namespace

[88] E.g. the proof-of-concept application called the *DistributedShell*

[89] Most popular is the iterative graph processing approach, based on *Bulk Synchronous Parallel* (BSP) model of distributed computation as introduced by Valiant in [134] and elaborated on multi-core in [135]. E.g Google's *Pregel* as described in [136] or its advanced open-source alternative Apache's *Giraph*.

[90] Current stable release 2.6.X also supports *High-Availability* (HA) feature that brings *NameNode* architecture that stores the directory tree for HDFS files and track data storage place in a cluster. This allows to build out horizontally, while creating multiple redundant NameNodes that share same data storage pool, thus scaling better. A *snapshot* functionality (v2.2) has also enabled backup and disaster recovery. Since v2.3 a heterogeneous storages in HDFS has been enabled (HDFS-2832) and due to v2.6 update it also supports APIs for using heterogeneous (also memory -HDFS-

management, and thus horizontal scaling, performance improvements, and multiple namespaces. It also eliminates *single point of failure* (SPoF), in the form of Hadoop v1 single node namespace management. This separation of HDFS (storage) from MR with YARN, made Hadoop the environment that is more suitable for operational (real-time) applications that can not wait for batch jobs to finish.

- *MapReduce* - Since Hadoop v0.23 the MapReduce (MR) has undergone a complete overhaul and resulted in *MapReduce 2.0* (MR2). MR2 has been designed to provide a more isolated and scalable model than its MR1 predecessor, due to no single resource management, scheduling and task monitoring work. This is achieved with each job in MR2 controlling itself, with its own *ApplicationMaster* (See Fig.2.19) administrating the execution flow of scheduling tasks, handling speculative execution and failures, etc. MR2 also has no more a single central *JobTracker* (See Figure 2.17(a)) responsible for resource management, scheduling and task monitoring work.

An application functioning in YARN based architecture is based on three general stages: submitting of the application by the client to the *ResourceManager* [91] , defining bootstraping application master instance [92] that will govern the actual application execution, and finally, the actual execution of the application managed by ApplicationMaster instance. The whole process can be described in eight sequential steps (also illustrated in the Figure 2.19):

1. Client sends the application and its launch (e.g. resource) specifications for `ApplicationMaster` to `ResourceManager`
2. `ResourceManager` determines the resource allocation - *Container* - on a specific host node to launch `ApplicationMaster`'s tasks. Next such `Container` is used to launch `ApplicationMaster` by `ResourceManager`.
3. Client is informed about the ApplicationMaster launch details, thus can communicate directly with its own ApplicationMaster.
4. `ApplicationMaster` uses resource-request protocol to negotiate more appropriate resource containers on other nodes.
5. Each of the remaining Containers is launched with `ApplicationMaster` by providing each `NodeManager` [93] their launch specifications.
6. Then the application-specific protocol enables sending each `Container`'s application status information to the `ApplicationMaster`.
7. The application-specific protocol also enables the application `Client` to receive from `ApplicationMaster` total status and progress data.
8. In the end when the application completes, `ApplicationMaster`'s jobs is to deregister itself with `ResourceManager` and free the resources for following resource requests of other cluster applications.

YARN usage statistics are not impressive [137], however, this is probably due to its recent [94] – as of 2015 – introduction and migration process safety complexity due to big data volumes and

5851) storage tiers by the applications (HDFS-5682). As for v 2.3 (HDFS-4949) there also has been Hive/Pig/Impala improvement for effective cluster memory management by dint of possibility to explicitly cache important datasets and placing their tasks for local memory.

[91] As a a contrast to its name, it is a pure-scheduler, as all of the resource (hostname, memory, CPU and in future disk/network/IO/GPUs etc.) fault-tolerance is moved to the *ApplicationMaster* (which makes it much better scalable). It redistributes cluster nodes' resources in the process of intermediate among resource competing applications.

[92] *ApplicationMaster* represents the instance of a framework-specific library per application in a cluster that also negotiates resources with the *ResourceManager* and execute/monitor resource consumption due to *NodeManager* communication. See Figure 2.19

[93] One-per-machine, responsible for creating applications containers, monitoring their resource usage and reporting it to the *ResourceManager*.

[94] Announced Aug. 2012 as an Apache Hadoop sub-project in the *Apache Software Foundation* (ASF)

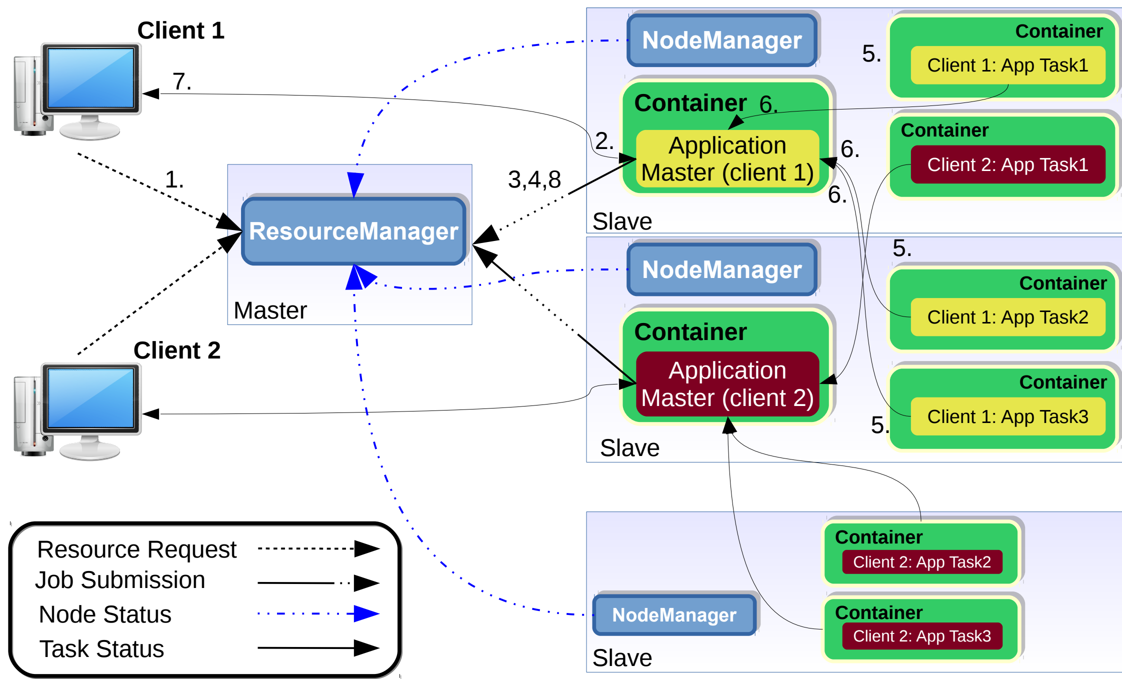


Figure 2.19: YARN based application lifecycle.

their significance. However, binary compatibility between Hadoop 1 and 2 has been maintained and new important features, as of present, are being implemented [138].

Stinger Initiative – Hive Strikes Back With Interactive SQL Sting The Impala approach of starting with the ground up as a new project, regardless of Hive has caused an interesting community effort, to preserve the investments of Hive's end users and broad ecosystem of vendors already integrated with Hive and to modernize Hive to be *real-time SQL ready*. This initiative initialized by Hortonworks [95] was called *Stinger* and turned out to be a success since 2013. There were three roadmap vectors for Stinger: Speed, Scale and SQL-compliance. The goal

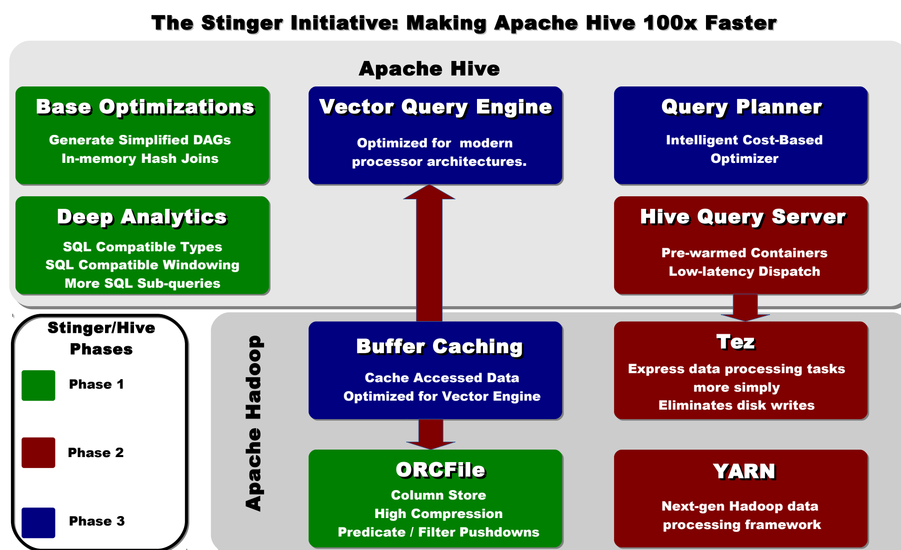


Figure 2.20: Stinger evolutionary steps.

[95] Yahoo's spin-off dedicated to driving the Apache Hadoop bus with Apache Hive.

was to assure batch and interactive SQL query workloads in a single engine. Hive used to process even simple queries with hundreds of records within minutes. This was due to startup overhead, un-optimized file formats, CPU-consuming inner loop, and file spooling costs. The YARN made it possible for multiple new engines to emerge for Hadoop, however the most popular Hadoop integration point still remained SQL/Hive. Joined community effort has not only proven its case but also has resulted in impressive performance boost. Stinger evolved the Hive's traditional architecture and made it faster, with richer SQL semantics (as of near SQL:2011 compliance) and petabyte scalability.

The Stinger Initiative has assumed a three phase (Figure 2.20) evolution (See Figure C.6 for details).

- Phase one - Hive v0.11 - introduced optimized ORCFile, data types made more SQL-compatible, analytic functions [96], aggregate [97] functions and making star joins more efficient.
- Second phase - Hive v0.12 - involved significant increase in SQL semantics [98].
- Third phase - realized in Hive v0.13 included a major speed and scale improvements with *Tez* integration. It has enabled executing queries on Tez, thus the dataflow model on a DAG of nodes facilitated more efficient and less complicated query plans for interactive queries to run on Hive. Also a Vectorized Query Execution has been included for better CPU computation optimization and - for the first time - a *Cost-Based Optimizer* (CBO) has been supported to generate efficient execution plans by examining the tables and conditions specified in the query for join reordering. The benchmarks has conformed the expected performance gains (Figure C.7; benchmark details [139]).

As already mentioned the Stinger initiative brought many improvements that were mainly possible by dint of the following projects:

- The *Tez* project - new data processing engine for Hadoop that has not only support for batch processing but also interactive data processing on a large scale. With emergence of YARN, Tez enabled multiple data access applications to work on the Hadoop petabytes of data over thousands of nodes while additionally enabling the users to express complex computations in the form of dataflow graphs with dynamic performance optimizations for specific data/resource requests.
- The new Hive's Vectorized Query engine allows to process more data in less time, thus improving scalability and enabling impressive CPU and throughput gains ()
- *Optimized Row Columnar* (ORC), a new file format providing high compression and high performance

Tez is an implementation of [140] paper describing the *Dryad* MapReduce generalization from Microsoft. Tez is an extensible framework targeted for building high performance batch and interactive data processing applications, coordinated by YARN in Apache Hadoop. It can be considered as a more flexible and powerful successor of MapReduce framework, however, it is based on expressing computations as a dataflow graph. Replacing MapReduce with the Tez application framework enabled a complex directed-acyclic-graph (DAG) of tasks for processing data. The Tez project became important – from the scalability point of view – as it allows to reduce multiple MapReduce jobs, required by Hive's jobs, into single Tez job. This made the scaling much more straightforward. Tez has also enabled Hive or Pig to run a complex DAG of tasks (Figure 2.22). For the first time Hive v0.13 has also given a preview of transactions by

[96] RANK, LEAD/LAG, ROW_NUMBER, FIRST_VALUE, LAST_VALUE, etc.
[97] OVER functions with PARTITION BY and ORDER BY
[98] Data types: VARCHAR and DATE and better performance of ORDER BY and GROUP BY
[99] Source: Hortonworks Inc. http://hortonworks.com/wp-content/uploads/2013/05/H1H2Tez-1024x537.png

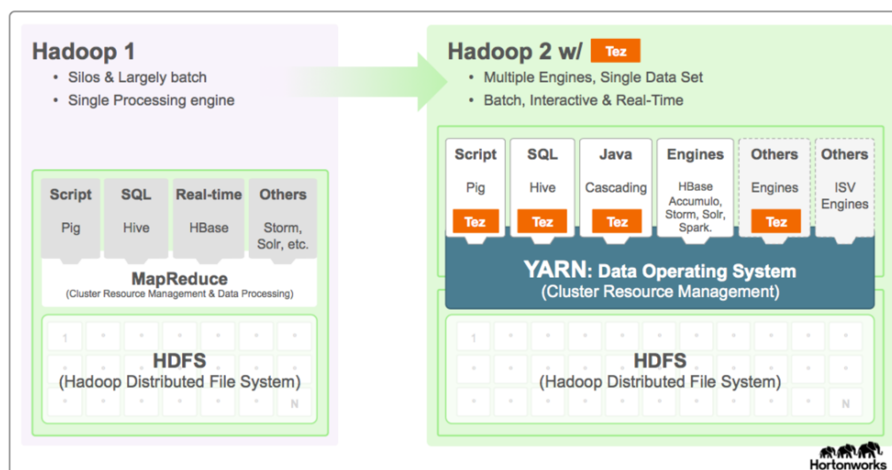


Figure 2.21: Tez API/framework to write native YARN applications[99]

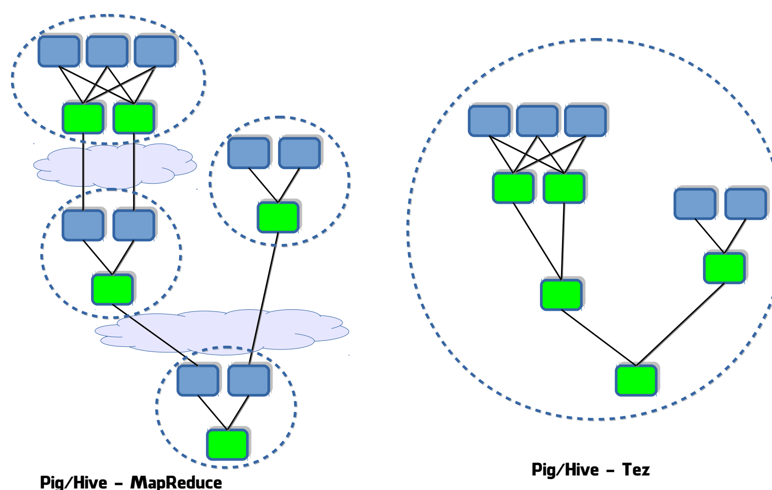


Figure 2.22: One Tez job instead multiple MapReduce jobs thanks to DAG

enabling data streams into Hive using Apache *Flume* [100] , making data available within seconds. ACID has also been considered in this version mainly for managing dimension tables and master data providing consistency and repeatable reads.

***Stinger.next*: ACID and Transactions in Hive** After the *Stinger:Initiative* has succeeded in making some of the queries faster a new version, called *Stinger.next*, introduced the goal to provide Hive with fully and truly near real-time responsiveness. It aimed at sub-second response time and required fast per-row query processing [101] and low query setup time [102] . The first phase of Stinger.next was released late 2014 in the form of Hive 0.14. Its goal was a sub-second query, interactive response times, transaction support, a full SQL:2011 analytics for Hive to allow rich reporting and CBO (see [141] for details) for complex queries and tool-generated queries to run on the Hadoop scale. The crucial - from the enterprise point of view - are the transactions and ACID compliance. Since Hive was first used for write-once, read-often applications with multiple

[100] Framework to aggregate huge amounts of data into the Hadoop environment in a distributed and parallel manner.

[101] Developed as a continuation of Tez integration and Vectorized Query Execution from Stinger:Initiative.

[102] This was developed with multi-threaded service process (daemon) - *Live Long and Process*(LLAP) working on every node. LLAP maintains an in-memory data cache, thus reduce process startup costs, I/O latency, and deserialization overhead.

partitions the ACID provided paradigm shift with *Insert*, *Update* or *Delete* SQL transactions [142]. However, at the time of writing this dissertation, according to [143] the work is still ongoing. However, there are still some interesting ideas proposed like integrating with replication tools for periodical (arbitrary time threshold) updating data from operational (OLTP) databases. The actual plans have partially become implemented according to the schedule in Figure 2.23 and are depicted in more detail in [144], however, due to not being fully implemented will not be discussed in this dissertation.

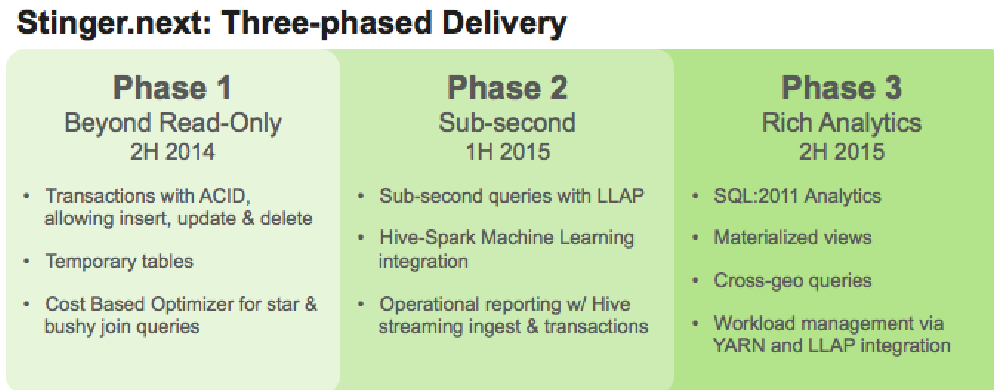


Figure 2.23: *Stinger.next* roadmap.

Tez vs Spark on Hive Spark has already been mentioned in section 2.4.3.3. It is a DAG executing project similar to Tez. Spark however, is more mature and has been developing for 5 years based on the Bekeley's [145] paper, that on the other hand, is built around the [140] which Tez is based on. It is worth mentioning that both frameworks provide:

- Distributed execution engine targeted towards processing large amounts of data that can handle arbitrary DAGs
- Use the MR I/O format to read/write from Hadoop

However, Tez is more focused on provision of faster engine than MR under the Hadoop's languages like Pig or Hive. Tez also provides API for DAG construction (edges,vertexes) and defines its dataflow.
On the other hand Spark also provides not only a better engine for Hive/Pig than MR, but also extensive API that makes code development much easier compared to Tez [103] . What is very important, Spark also provides *Resilient Distributed Datasets* (RDDs) abstraction which is extremely useful to process distributed in-memory data. Spark also tends to be supported by multiple companies such as Intel, Yahoo! or Cloudera whereas Tez is strictly one company - Hortoworks. Therefore, Tez seems to be most suitable for backend of Hive/Pig based execution engine over MR. Spark, on the other hand, tends to be better suited for direct API usage, writing data transformation job, implementing a distributed machine learning algorithm or with its own and dedicated high-level data processing language.

⋆ ⋆ ⋆

Apart from that there are also other less popular frameworks like Presto (Facebook) or Lingual based on Cascading API. To compare the discussed cutting edge technologies, a dedicated benchmark was developed at AMPLab – UC Berkeley [146]. This benchmark is based on workloads and queries from already cited [125]. It includes Redshift, Hive, Shark, Impala and Stinger/Tez comparison. The latest results (February 2014) show that still Impala and Shark outperform Hive, but only 3-4x.

[103] *WordCount* code for Tez is approx. 300 lines where for Shark is only 3.

	Hive	Shark port of Hive -deprecated	SparkSQL	Impala	Lingual
Engine	Hadoop / MapReduce or Tez (faster)	Spark		C++ custom replacement of Hadoop	Cascading on top of Hadoop or other
SQL dialect	HiveQL	HiveQL	HiveQL with Catalyst optimizer	HiveQL	ANSI SQL
Developed By	Facebook	Bekeley AMPLab		Cloudera Inc.	Concurrent, Inc.
Performance	Low for real time	30x faster than MapReduce		100x better than MapReduce	-
Supports	Textfile, SequenceFile, ORC, RCFILE, Parquet	Native JSON & Parquet	JSON, Parquet, Hive	Text, Avro, Parquet, RCFile, and SequenceFile.	Text, csv

Figure 2.24: Compute engines

2.4.4 Enterprise Service Bus (ESB)

The integration processes are often more than only a dedicated and even distributed heterogeneous environment. Every integration architecture must also consider the networking model of communication between the storages in the distributed system architectures.

Since early beginnings of data integration the basic mechanism of integration is batch file transfer. It does live on in modern systems as it is not effective but also extremely simple that many designers use it, and thus it has become almost impossible to be eradicated.

***Enterprise Application Integration* (EAI) hub** The more advanced architecture considering integration is the *Enterprise Application Integration* (EAI) hub. It is based on single central system that the applications from the integrated sites must be used to communicate. It must understand all the different formats, transform them between each other, and thus links the different systems together. Its big advantage is reducing the overhead of P2P connections from n^2 to $(n-1)n/2$, where n is the number of sites. This model is often present even in modern ESB solutions. The EAI hub advantages are easy to mange central operating point and well understood pattern. On the other hand, we get a single point of failure, and thus it becomes problematic to scale it. What is more, every party involved in the hub processing must strictly conform the settled communication protocols that require all participants (possibly from distant large companies) to meet and settle every single change. Moreover, one serious threat is that the data flows in the EAI hub will be additionally accompanied with hub-resident business logic which would change a simple and lightweight integration solution into a messy and unmanageable one, especially when considering scaling-up.

***Message Oriented Middleware* (MOM)** The next step in progress of integrating solutions evolution was the *Message Oriented Middleware* (MOM). It is largely similar system to the mail system that would be brought to computer messaging. It is based on decoupling message producers from consumers in location-addressing and in time. This involves sending one-way *asynchronous* messages to a queue, rather to particular consumer; with request replies using reply queues. This means that the request-response can be done but not in an instantaneous manner. The MOM is often referred to when considering reliable delivery. What differs MOM

from another model of SOA is *message decoupling*. In MOM the inherent assumption is that both sides understand the same message format as a part of core model. A crucial aspect of MOM is the transaction processing based on queues. It is often misconducted that transaction is a *distributed two-phase commit* (DTPC) where all actions are part of the same transaction. However, the in real world there are not many DTPCs because they tend to be ineffective. It ties up all the distributed systems in a voting process. Thus the overall requirement for all systems to be fast and efficient becomes problematic. In MOM we deal with a queue broker and a collocated database, that is with fast connectivity and high availability. Thus one can proceed with updating the database, and enqueuing of the message in the same, local transaction in a fire-and-forget way, as the queuing system takes care of the rest. The other side accepts the message and puts it into its queue message broker, then applies the changes in its transaction locally. This can be done without any real distribution. That is why the reliable messaging is important. However this model is not a distributed-transaction-based because we have two distinct transactions. In the case receiving party's transaction fails, the sender will not roll-back. In such a case compensation is required. This means that the site with the failed transaction has to send a message to the queue that its transaction failed and that the sender must roll-back. The advantage over DTPC is that the transactions happen very fast, asynchronously and do not tie up the entire system.

The most significant protocol for MOM is *Advanced Message Queuing Protocol* (AMQP) – driven from the financial services and also used by cloud environments. AMQP was designed starting with taking SMTP and applying for machine-to-machine messaging. This protocol is crucial because most of the MOM vendors (WebSphere SI, TIBCO, Sonic) provide their own proprietary protocols for MOM. Thus the MOM advantage of interoperation on the application level, is that it is tied strictly to one vendor infrastructure.

***Event Driven Architecture* (EDA)** The EDA is an evolutionary fork of MOM. The EDA moves the MOM's consumer-producer connectivity decoupling even further, by publishing an event as a topic while many consumers may subscribe to this topic. Actually, this is a straightforward analogy to mailing lists and the way how Apache Foundation works. Its advantage is that while creating a topic, it does not have to be known of how many and who will join it; thus, subscribing to the event after a fact has taken place. In contrast to EAI, it also eliminates the need for collaboration netting that settles the communication rules in order to plug into the EAI hub.
The problem with EDA is when systems event feedback loops might occur. For example two parties creating events when something changes, and then signal both to update state, thus there is no way to know if the event is done by the party or induced by the change that again creates an event etc. The solution, however, to this, is creating a *master data repository* gathering all of the events and re-publishing only those that were new and/or wanted.

***Service Oriented Architecture* (SOA)** We could state that SOA directly comes out of XML. The fact is that understanding the schema and the structure of the messages and having the metadata, policies and security is the key discrimination between SOA and MOM. The security part came from the fact, that with present solutions there can be no assumption that the integration is going to take place within a company's intranet, and thus must be considered along the development of the SOA implementation.
The point of SOA architecture is that it is a service, thus one can use a service without all the concerns of possessing it. Therefore every SOA architecture is justified if there are more service consumers per service than one (EBay has a rule of at least three consumers per service). Thus SOA is not to be implemented as a client-server architecture e.g. using XML. On the other hand, the ESB should not be misconcepted with the SOA implementation. It does not guarantee the proper SOA model.
An example of a real, complex and distributed integration SOA is EBay, with very lightweight, policy-based mediation that monitors and routes messages to their destination. Moreover, it

has high reuse with simple system assurance for common message schemas and formats that results in high productivity. EBay has actually open sourced its complete framework for SOA called *Turmeric*. Another example is Amazon that moved to SOA back in 2006 with multiple service components with common interfaces. Also Netflix uses the SOA model based on the REST calls which are in turn based on Amazon cloud architecture.

2.4.5 ESB / SOA - Rules of Engagement

The main characteristics of a proper ESB should be based on *policies*. Policies should be out of the code. One could use e.g. *eXtensible Access Control Markup Language* (XACML) [147] standard as an example of extracting authentication, throttling [104] and *Service Level Agreement* (SLA) [105] policies from the code. Moreover, such architecture would require an independent management based on loose coupling of configuration. This provides the abilities for hot deploys / re-deploys in a continuous delivery manner. On the other hand, the continuous delivery requires the knowledge of how and who will become affected by each new change, and thus an exact information of dependencies and entire lifecycle is crucial. Additionally, the analysis and reporting of the metamodel; e.g. to find out (possibly by querying) who had been granted access to a particular operation according to policies, is important. Now the analysis and reporting, so as lifecycle and dependency management is essential for well designed *governance*.
Finally, the most basic but at the same time important is the non-blocking, asynchronous routing of the messages and the distributed nature of possibly multiple ESBs inter-communicating in distributed architecture.

The true SOA-based ESB is all about moving from *producer-to-one-consumer* (i.e. client-server) into *provider-to-consumers*. The Internet changes everything but what is really working for this kind of environment are the services and the APIs. Services are a middle layer between the business data and the open APIs to heterogeneous consumers – i.e. the use of all known platforms, tools, and languages. Now what is a key aspect of SOA is that it moves from traditional

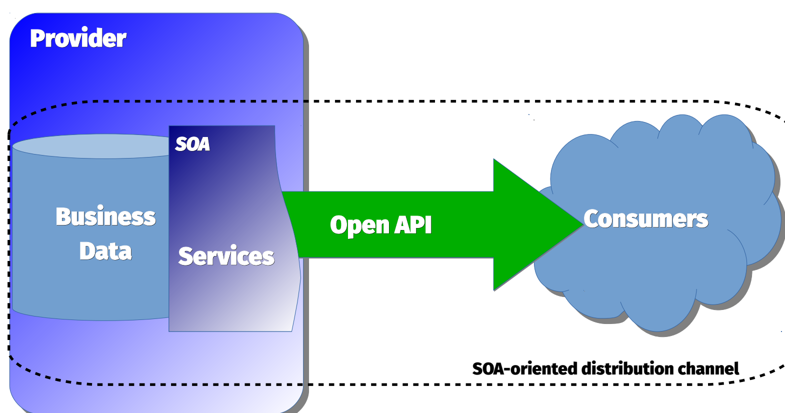


Figure 2.25: Service-oriented data distribution.

enterprise service-oriented architecture to a more web-like open API model. However, API is not just about a message format or the message model. To move from one consumer to multiple heterogeneous consumers, API should consider:

- *simple licence* – allow others to contribute and reuse the API within their new products

[104] E.g an integration throttling may be embedded in application logic to prevent overloading and slow end-system down, in an expected way, thus slow down the publishing service in the acceptable manner.

[105] Service Level Agreement formally defines service in aspects of scope, quality and its responsibilities that is agreed between the service consumer and provider.

- *the ease of use* – services consumable from any platform, tool, or programming language – e.g. REST
- *reporting and billing* – understandable cost policy and traceable spendings
- *account management* – use detailed client tracking and assure API's customer classes
- *self-service* – automated and online singn-ups, licensing, testing, support and integration, enable fast and cost-effective multiuser interactions (e.g. Salesforce, Amazon, Google handle hundreds of thousands of customers)
- *developed community* – design APIs not only to satisfy current customer need but also to enable developers to make custom changes based on it in the future, due to providing them with tools and documentation of how additional functionalities can be handled

– as the main aspects [148] while being designed. In the API's design the remaining two aspects that must be considered in order to write Web environment ready solution are governance – due to rate limiting, security audits and version management – and the SLA. The correct design of API will provide the view of how the API is used for both: the consumer and the provider. A regular, core ESB does not provide that to gain such functionality an additional SOA management system is required.

DEFINITION 2.15: (API)

API is a business capability delivered over the Internet to internal or external consumers. It consists of network accessible function, available using standard web protocols with well defined interfaces and designed for access by third-parites

To distinguish API from service the API provider assumes that it is ready to use out-of-the-box without prior configuration / meetings and settlements. Moreover, it should be easy to consume. In the API sense *governance* must care about consumer specific needs and constraints. While considering API maintenance changes, the backward compatibility of crucial elements should consider it and conform to the SLA and analytics. There are already very well suited tools to achieve this like OAuth2, REST / Swagger technical and readable description models,

Concluding one should consider APIs and services as policy and metdata -based with a configurable way to build governance in a very programmatic way.

2.5 Conclusions

Databases should leverage the fact that the hardware and networking architectures have changed substantially since the databases were first invented. The emergence of multiple dedicated data sources has caused the proliferation of integration and processing of heterogeneous data. However, regardless of being a classical data warehouse or a distributed analytical BI tool each and every solution require2 data to be extracted and loaded to the external data storage. In the case of data warehouses the ETL tools enable this job while in the case of Hadoop alike ecosystems we also need tools such as Apache *Sqoop* [149] that will transfer data from operational sources. Such tools might work in an incremental manner or free form SQL queries. Moreover, they might also provide means to export the data back into operational sources from the cluster. However, one way or another, this requires the operation of data transfer – and in the case of Big Data it becomes a serious issue. While most of the integration problems, already discussed in 2.2.6, has been challenged and overcome - there are still unapproached issues of processing most current data. This is not only problematic in terms of amount of the data, but also its validity with constantly changing data at the data source site. In the following chapter the author introduces a new integration architecture based on the dedicated metamodel that will by-design force usage of the most current state of the data.

CHAPTER

3 The Model of the Architecture

“For abstraction consists only in separating the perceptible qualities of bodies, either from other qualities, or from the bodies to which they apply. Errors arise when this separation is poorly done or wrongly applied: poorly done in philosophical questions, and wrongly applied in physical and mathematical questions. An almost sure way to err in philosophy is to fail to simplify enough the objects under study; and an infallible way to obtain defective results in physics and mathematics is to view the objects as less composite than they are."

— *Denis Diderot, A Letter on the Blind for the Benefit of Those Who Can See, 1749*

“The key to growth is the introduction of higher dimensions of consciousness into our awareness."

— *Lao Tzu*

The integration is a very wide and prominent aspect of modern data systems – especially enterprise ones. Systems incorporating hundreds or thousands of data centric, custom-build applications of various origin, operating in numerous tiers on multiple operation systems and platforms of dedicated or cloud environments, represent a complex multilevel communication network. Therefore, integration is not an easy task. The enterprise solutions are based on heavy, dedicated, vendor provided *Enterprise Application Integration* (EAI) hubs or *Enterprise Service Buses* (ESB). Despite urging needs only a few standards made it to establish themselves in this area. The XML or XSL and Web Services give a lot of opportunities, however, they also introduce a new ground for different incarnations and interpretations of those standards. Thus, there is a serious interoperability gap between, even the standard-based integration solutions. Even such sophisticated integration solutions as the OMG’s CORBA [87] had problems with assuring inter-system integration for non-interoperability parties. The XML, is often referred to as a *lingua franca* of data integration. This might lead to wrong suggestion that the XML and XML-based Web Services can solve the system integration issues completely. This is mainly due to the fact that having a common alphabet does not mean that what one has written in one language is understandable to all languages that use the same alphabet. Thus the semantics is the real obstacle. As already discussed in the previous chapter, covering the semantic heterogeneity is a very time-consuming and complex task, that is, however, crucial for correct and well-defined integration. This problem will be approached within this chapter, along with the most basic architectural considerations and the discussion proposed by the dissertation dedicated solution.

3.1 Data vs Application Integration Patterns

One should be aware that there is no simple universal, golden rule for integration, regardless if we speak of data or application integration. Object -oriented design, Service Oriented Architectures, Event Driven Architecture, Message Oriented Messaging are only means to reach a specific goal. This goal however, is always somehow different and requires a dedicated effort.

A well known concept from the software engineering in the form of the design patterns will be helpful to provide some specific establishments on the ground of this dissertation. The discussed meaning of the *pattern* term was first introduced in the field of building architecture in [150] by Christopher Alexander[1] . What he has noted is that often general problems have repetitive sub-problems, that in turn, can be approached in the same way – while still being a part of a bigger problem, and what is more, possibly contained within different general issues. Therefore the pattern is a general, reusable solution to a commonly occurring problem. Because the patterns are discovered (rather than invented) facts of life, it has become straightforward to apply patterns in computer science, in the form of good practice formulas. With the time, the design patterns have particularly immersed into software engineering and also software architectural design. We now consider patterns as smart formulas that describe solutions to frequently recurring problems also in terms of data integration.

3.1.1 Patterns in Software Development

The periodically occurring problems in software development can be discriminated into two classes, namely the software design patterns, and a broader scoped – the architectural patterns. In the design patterns area, a design motif is described by a structure, actors and workflow rules. Now each design pattern can be applied, according to those three factors to implement prototypical micro-architecture that will adopt particular design pattern to developers recurrent problem. The developer's choice however, is to pick the pattern that will suit and optimize the problem implementation effectively by making the problem micro-architecture solution similar to the appropriate design motif. This transformation of a design pattern into the source code is a form of a good practice in software development, however, it must be made with attention and applied only when undoubtedly required. The design patterns as a part of software craftsmanship are going to be considered in this dissertation prototype implementation, however it is not its clue or main area of interest and therefore it will not be extensively discussed.
A wider scope of the pattern appliance is considered with an architectural approach. Just as in the previous class, the architectural patterns in software development are considered in terms of good practice solutions for solving commonly encountered problems. Apart from detailed software characteristic issues (as in the case of software design patterns), the architectural pattern should also consider performance limitation, software network particularities, availability context and business circumstances. With such extensive expectations, architectural design patterns not only tend to be a complex task, but also often become a crucial part for every enterprise-class software architecture.

DEFINITION 3.1: *Architectural Pattern*

Architectural pattern assembles a cohesive image of the system that depicts important system components' collaborative nature. Each architectural pattern can be reused for multi-domain appliances – as a architectural pattern – as long as it is applicable and strictly defined.

Therefore what is truly considered – is a set of architecture design decisions that must fit within the context of specific system and provide actual profits from applying, in terms of e.g perfor-

[1] He proposed the use of collections of architectural patterns to address deficiencies in modern building design.

mance, ease of management, clean code etc. The domains that use the modern architectural patterns have already been discussed in Chapter 2 e.g. the data architecture (*data warehouses, data marts, transactional data stores - OLTP* architectural patterns), Business Intelligence (*transactional/operational/analytical reporting* architectural patterns), Master Data Management [2] , Data Modelling (*Entity-Relationship model* [3] *, dimensional data modelling* [4] architectural patterns), etc. One last area would be the *Data Integration*. It is mostly considered in terms of *ETL* and *Managed File Transfer* (MFT) [5] (patterns such as SOA, B2B, Cloud, Ad Hoc, etc.). However, there are still two more patterns of EAI and ESB that we have already discussed in general in the previous chapter but they will be elaborated more in the following section 3.1.2, in the context of the dissertation proposed architectural pattern.

One indisputable benefit of using an architectural pattern – that needs to be mentioned – is the provided language and vocabulary unification that is about to be used across the entire problem solution. By this, we can already see that the semantic integration must always be present, while considering architectural patterns. It brings us to the idea of architectural pattern considered in the scope of integration.

3.1.2 Architectural Patterns in Integration

Semantic unification is just a single example of how the architectural pattern can apply for the sake of data integration solutions, and to be more specific – how the architectural pattern can bring profits for the architecture presented by this thesis. Architectural patterns play the most general and significant role on the ground of integration by filling-in the gap between the integrated system implementations and the integration abstraction.

The approaches considering general integration have to be discriminated between two conceptual levels, or as it will be used – patterns. Firstly, the integration can be considered in scenarios where multiple applications need to communicate and transfer transactions. This is the case of higher level system integration – eg. EAI/ESB. The second scenario involves integration on a more basic level i.e. the data level. Both integration scenarios consider the same goal of providing a unified and homogeneous interface to the system data and/or analytics.

EAI In the first – application integration – pattern, the level of complexity tends to be more challenging. This is due to the fact that each application owns its specific particularities. These particularities are only considered within such a specific application and include aspects of the data storing mechanisms, geographical dispersion, configuration, interface language, platform, operating system, access rights, user management, etc. – that are different across multiple applications. Therefore, each application has its own general assumptions (often with a very high detail level) that can not be foreseen by the designer of application integration solution at the moment of its creation. This has led to a specific pattern for the application integration that assumes no interference with the application of internal mechanisms. It states that the integration should be limited only to the interactions with each application on a *black box* basis, while only participating in the integrations based on the *messaging* interface communication. This gave a start to a large number of *Enterprise Integration Patterns* (EIP) [152] that has become incorporated into multiple EAI solutions.

[2] I.e. single-point-of-reference for processes, governance, policies etc.

[3] Typically implemented as a database. Can be expressed e.g. with the UML class diagram.

[4] According to [107]: design technique for databases aimed at supporting client queries in a data warehouse. It is oriented around understandability and performance. Based on the concepts of facts (measures), and dimensions (context) with star-like schema. See 2.4.1.1

[5] Successor of FTP that in contrast can handle adequately support secure, automated, managed and audited file transfers and according to Gartner Inc. [151] is a recommended for organizations replacement of FTP.

> Unrestricted sharing of data and business processes among any connected application or data sources in the enterprise.
>
> *– Gartner Group, Inc. on EAI (see [153])*

EAI is composed of a set of technologies and services used to compose a middleware across the integrated systems that would not be able to communicate otherwise. EAI main target is to simplify the process of business/analytical tasks without involving any, or almost any effort from the integrated applications [6] . In general the EAI is about sharing applications data and business processes. Main appliance for EAI is often the data integration in form of the Enterprise Information Integration (EII) (see 2.2.3.1), business policies and rules generalization (i.e. vendor transparency), and applications' common, unified access interface.
EAI mostly uses mediation (for inner mechanisms) and/or federation (for outside client request service) based patterns (see 2.2.4). Those patterns act respectively: as a syncing broker between integrated applications that propagate a new event across the remaining applications or as a transparent interfacing facade for client calls. Within those inner / outer patterns one can also easily find some of the synchronous/asynchronous communication patterns. As already discussed in section 2.4.4 the EAI can be realized with the *hub-* or *bus- based* pattern. Both being designed for specific purposes.
In most cases it is true that each EAI solution should include:

- central broking facility – that handles policies, coordinates EAI-provided services and communication e.g. with ESB solution
- canonical data model – used to communicate across the integrated system
- mediator/wrapper – a facility that will enable a single integrated component to communicate with the centralized broker
- API model – for the standardized way of communication with the integrated system from the outside client perspective

All key middleware technologies required by such EAI system in the form of MOM, SOA, EDA etc. has already been discussed in section 2.4.4. The central, broking facility and the entire integrated system uses a canonical data model – most commonly – through an Enterprise Service Bus (ESB) pattern based or a hub based solution for communication. The communication must consider message routing, mediation, event processing, security, policies, and many more. Now an ESB is evolutionarily more advanced than the hub model. A rule of thumb for a well defined SOA-based ESB has already been discussed in 2.4.5. However – as of the present – there is no enterprise or global standard for ESB. It has evolved from MOM and became an implementation standard for the SOA-based architectures. While the current ESB technologies are the EDA and MOM solutions with combination of message queuing, some of the vendor-claimed-to-be ESB solutions still do not implement it in the discussed way. This is because ESB should be platform-agnostic and possess the ability to integrate with anything under any condition. Some vendors limit the ESB only to their own solutions.
Undoubtedly, ESB based EAI brings the benefits of scalability and, no coding in favour of configuration. Also when used in the non-centralized architecture it eliminates single-point-of-failure, and a loosely coupled system for plug-in possibilities.

Issues of EAI/ESB On the other hand, some of the pros and cons characteristics of the EAI solutions have already been mentioned in section 2.4.4, however additional and the biggest flaw of the EAI integration is that it increases coupling between the integrated systems. By this

[6] Especially important when considering applications that for security or lack of support reasons can not be modified.

it increases the management costs and brings additional overhead for infrastructure maintenance. The other disadvantages would be its complexity of management, competing standards, or constant collaboration and settlement meetings (as of the hub based EAI) that lowers the effectiveness.

Data Integration A more in-depth pattern for integration is focused on the application target – i.e. the data. Some general discussion of data integration has already been conducted in section 2.2.3, however here we will abstract the most vital characteristics that will later be covered in the presented solution.
Each application goal is to process its target data in a dedicated manner for a specific purpose. While it is enough for a single application site, however when the integration of multiple application sites of various origin is involved, it brings a need to provide translation between each application from each site. As already mentioned in the first scenario of the application integration, the goal is then to integrate communication between applications. Each of such applications plays the role of interface for their data or provide data-based analytical reports. Such integration goal is to provide separate applications to cooperate and produce a unified set of functionalities. While this is acceptable when the use-case scenario assumes data access by application interface, it becomes redundant otherwise. In those remaining cases one can exclude the application layer – with its programmatic characteristics burden – from the integration process, and provide direct data integration by eliminating all the unnecessary applications overhead.

As each enterprise-class system is designed for data processing, the integration of application would not have any further sense if the data available to multiple applications were available in a unified way to any application. If it were possible to develop a single view of the data, previously used by multiple applications on the distributed sites, it would eliminate need for EAI at all.

⋆ ⋆ ⋆

This is what the dissertation topic architecture is aiming at. The goal of this thesis is exactly to provide a general purpose data integrating architecture that would provide a virtual view of the integrated data. The data is made available due to the arbitrary global schema configuration, regardless of the data prior purpose, location or access characteristics.
The architecture enables a single client application to grant a direct access to any data that would otherwise require interface in the form of dedicated, local application. Thus, it eliminates the need for the application to collaborate with any other application while accessing the local data. The architecture, however, due to it flexibility and open-endedness does not eliminate the EAI possibilities. The EAI can still be applied and implemented to compliment the data integration possibilities. However, this is out of scope of this dissertation and is a target for future research.

3.2 General Architecture and Assumptions

Regardless of what kind of accustomed architecture pattern for integration we choose ETL, MFT or EAI/ESB, there are always some disadvantages that result in pattern specific appliance. It has become clear that there is no standard or even a pattern that could be general enough to overcome the disadvantages of the mentioned patterns and at the same time provide the same functionality, or at least anticipate its implementation due to its abstract and universal model. This is what has been aimed at by the presented research.

3.2.1 Virtualization as the Key to Integration – Postulates

The virtualization is a concept introduced early in the 1960s in the form of paging technique used for memory virtualization. Since then it has become widely used to virtualize almost everything

from hardware to operating systems.

DEFINITION 3.2: ***Data virtualization***

Data virtualization is the process of offering data consumers a data access interface that hides the technical aspects of stored data, such as location, storage structure, API, access language, and storage technology.

The introduced data integration solution, requires a high level of abstraction that impose the data to be considered *virtual*. This is due to the fact that generalization of a real (as in contrast to virtual) data solution would have never led to satisfactory outcome as it would be impossible to foresee all possible real data configurations and participation scenarios. Moreover, such an approach enables easy data management without the need of moving potentially large volumes of target data. The data manipulation is then more flexible, lightweight and stripped off the access technical characteristics, thus it can be formed and customized arbitrarily. Moreover – unlike ETL – the actual data is assumed to be fetched in a lazy manner and due to *as-needed* access approach, the data tends to be available in a real-time perspective. On the other hand – with contrast to data federation – virtualization eliminates the need for common data model across the heterogeneous data sources. However, a kind of canonical model is required. The context of connecting different data sources and accessing their data must consider logical, common interface and be prepared with custom transformation rules of their native access methods. What is more, a federation aspect of the proposed solution is present, as the resulting virtual data perspectives are based on data from multiple source systems. Finally, the data delivery is assumed to be available in the form of unified API provided for client calls.

The means of data virtualization are diverse – from the federation server exposed as a one store, through an ESB that plays the role of abstraction layer providing services for accessing the connected data sources, to cloud storage with API access and in-memory loading of the data from physical databases. In the proposed solution we will devise a kind of hybrid of those approaches, that targets to avoid most of their flaws.

As far as the scope of this dissertation is considered, the access methods are going to be limited to read-only mode without assuring transactions, however, those issues are recalled as possible to be implemented within the presented solution.

The benefits of such solution include reducing:

- data error risk,
- system workload (due to no real data manipulation)
- development and maintenance complexity
- data view reduced storage space (due to metadata based descriptions)

On the other hand, such an architecture might also impose some possible difficulties:

- Multiple and unexpected user queries without dedicated optimization can slow down the entire integrated environment
- Problems with business understanding/interpretation of the data
- Defining uniform application of business rules – i.e. the governance
- Lack of historic record of the data snapshots (with contrast to data warehouses)
- API change would require all consumers to be informed and comply

In the following sections those potentially problematic areas will be challenged with the newly presented approach (especially in section 3.4).

3.2.1.1 Virtualization – Existing Solutions

Existing solutions for data virtualizations are all commercially supported products such as: *Red Hat JBoss Data Virtualization, Informatica Data Integration, IBM InfoSphere Server* – i.e. incremental (data replication), and virtual (federated), *Oracle Data Integrator* etc. However, they are all closed source and provider support dependent. It is also not disclosed how the integration takes place in detail, possibly for commercial significance.

3.2.2 Polyglot Persistence – building "*The Tower of Babel*"

The integration solution that is discussed aims at providing an architecture for flexible data access regardless of its structural or technical dependencies. As each database is designed to solve different problems, the use of a single DBMS model for a complex enterprise environment is not desired. Storing business transactions, session management data, BI data warehousing, logging or reporting in same database engine is a mistake. Let us assume we have a common selling use case. It would almost always include a *shopping cart* and *session* objects. Now, each of those parts of the system would have different requirements towards storing properties. that is session management must be more available than e.g. order history, which in contrast would require secure backup replication possibilities. Thus taking the best of every of the available data storing engines seems obvious and is referred to as a *polyglot persistence*. As a consequence, one should consider using – e.g. key-value store for storing transient session or shopping cart content (until it is committed), as the RDBMS is not accustomed to handle transient data as effectively and responsively. Referring by the user ID key, in case of transient shopping cart or session, it is more natural for the key-value model to be used. However, after being transformed from the shopping cart into the final order, the actual data can be moved in secure and stable RDBMS. Next, based on this example, one can additionally connote recommendations for each product e.g. of "*what has been bought by other buyers of the particular item*" or "*what accessories are available for a particular product*". For processing such a graph based relations an obvious choice seems to be the graph based model DBMS. Moreover, one could refer to each *order* as a complex structure document of multiple items with their possible characteristics. Thus, moving such a complex structure to document store, while storing only the inventory and items price in RDBMS, would seem natural. The appliance of multiple data models is not only useful while

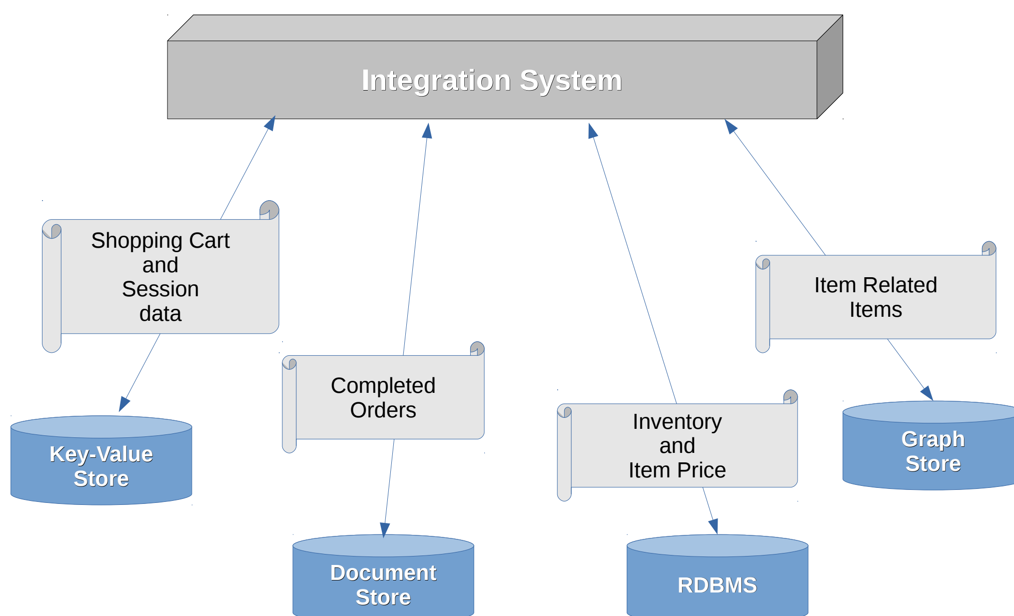


Figure 3.1: Polyglot persistence.

serving the prior goal of sales application. It could be also useful to provide additional analytical or reporting functionalities e.g. by using the graph stored data for market analytical purposes and client targeting.
To achieve this goal, one could also make the polyglot solution even more productive by providing services API for the graph engine, so that the graph relations can be accessed by multiple other applications, and not just the selling one that generates its data thus, providing more possibilities. The next step obviously is wrapping of each and every data source with a service with a defined API. Again, this allows interoperation with other applications without the need of changing the applications, and allowing for communication unification even in case of database changes. Additionally, the polyglot persistence operating – regardless if it uses multiple data

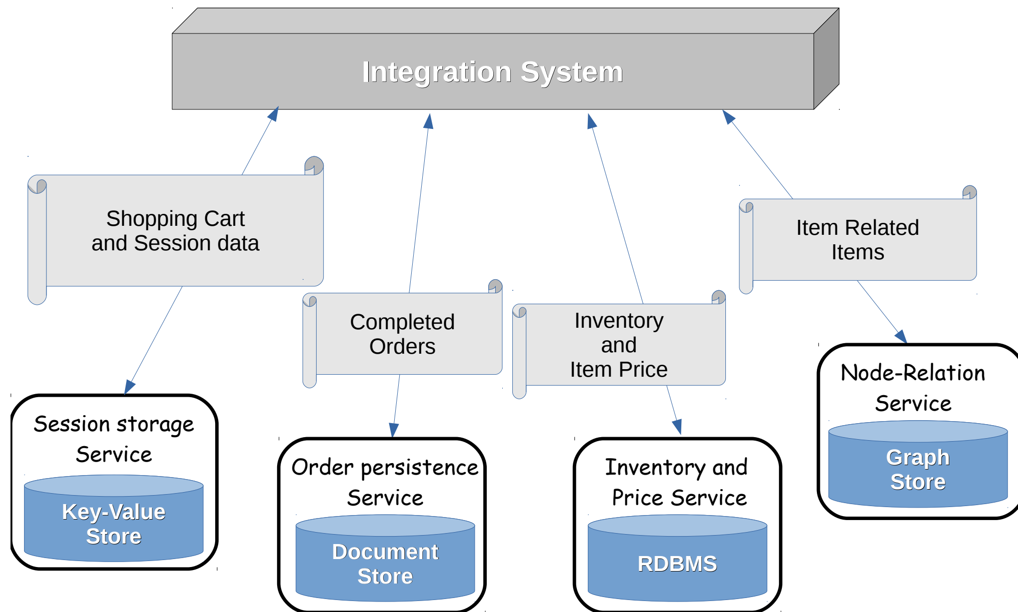


Figure 3.2: Service based polyglot persistence.

sources or not – can also be improved in terms of caching for better performance or indexing [7] for better search efficiency. It is especially important in the case when data storage can not be changed for specific usage due to existing legacy application depending on the existing data storage. The condition for such improvement however, is to comply to the data synchronization between the data sources and the cache/index engine. In other words, the index data must represent the current state of the actual data stored in the database. This can be done in a batch or real-time manner however, it must assure stale data in the index/cache. For the purpose of keeping index up-to-date, a special pattern is applicable in the form of *Event Sourcing* (see 3.2.3). As not all heterogeneous data sources can provide signalizing its data updates, in such cases every data source interaction would have to use the data source wrapped access service. Optionally one could develop a service that could check the data changes at the data source. [8] . However, in the case of the data source such as a flat file or a spreadsheet, this would have been neither an easy nor lightweight task.

As already described, one can obtain significant gains in performance while considering such an architectural pattern in terms of polyglot persistence. It is crucial, however, to pick the right technology for the right purpose. One could use the RDBMS for searching natural graph hierarchy of product relationships with hierarchical table structure and recursive queries but

[7] E.g. *Apache Solr* as a high performance search server for indexing and caching.
[8] This can be based e.g. in MySQL (MyISAM) on `information_schema.tables` database with its `UPDATE_TIME` or the use of triggers in other possible cases.

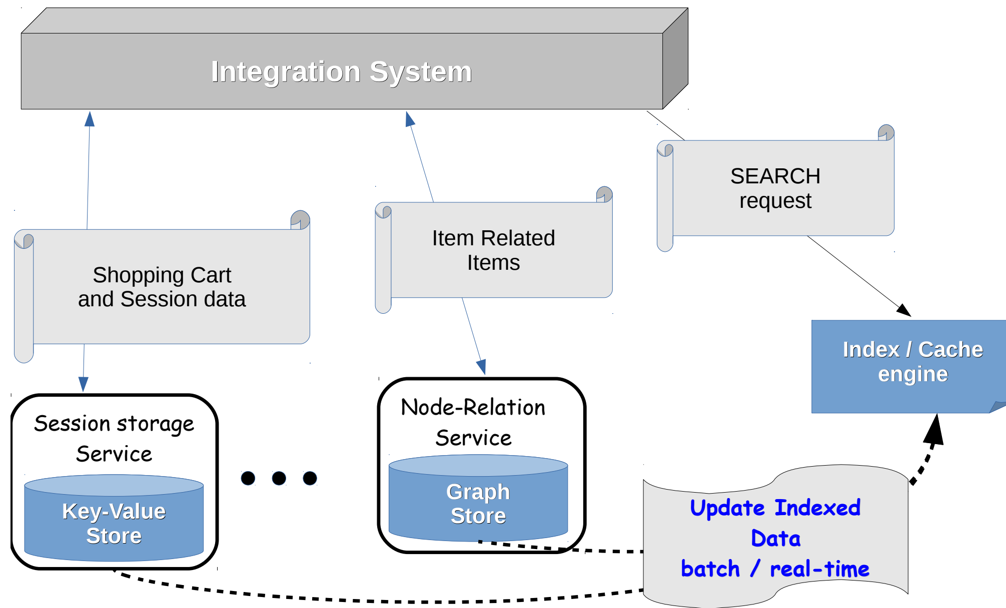


Figure 3.3: Service-based index optimized polyglot persistence.

this – as proved in Chapter 4 – is a way less efficient than providing a dedicated data model with all its benefits of node-relation based model.

3.2.2.1 Polyglot Persistence Concerns

As already mentioned in Statement 1 one has to use the right technology, for a particular purpose. To be precise, most of the data stores can handle defined data requests somehow by persisting the correct attributes. However, in the case the request changes, later it might be impossible to handle the modified queries when the conditions change. For instance, the relational database can handle a particular hierarchical query while the tables are modelled accordingly. However, in the exemplary use case scenario of "*recommended accessories*" or "*other buyers bought also*" the traversal nature changes between those two requests. In the case of changes of how traversal is done, a relational database model would require a refactoring and/or migration of the data to face the new data requirements. Now as for a polyglot persistence solution, that tracks relations between nodes in the form of metamodel, it is easy to simply programme a new relation model while using same data store and bring only changes to the metamodel, or use a minimal replica on a suitable store model (see chapter 4.2.4), in this case a graph model.

However, the polyglot persistence model brings in the complexity of management and security, which is additionally complicated due to some solutions being open-sourced, and thus often equipped only with community support. Although, there are some companies providing the commercial support, it provides an additional instability risk. Following the enterprise point of view, the proposed polyglot solution clients in the form of ETL tools would want to use the solution raw data. To obtain this goal a well designed API is a must [9] .

What one should especially note about the polyglot approach key points, is making the data access encapsulated with the unified API which reduces the impact of data storage choices on the client calls.

Additionally the polyglot solution proposed in this dissertation will provide a common and lightweight means for introduction of policies for easy data source integration and interpretation.

[9] The API is also provided within the solution discussed in this dissertation.

3.2.3 Event Sourcing as a Persistence Technique

A specific approach not only for the polyglot persistence, but persistence in general is formulated with the *Event Sourcing* as an architectural pattern. The approach is based on persisting all of the changes to the persistent state, instead of focusing on persisting the application state itself [154]. The idea is already well accustomed in the form of transactional logs of RDBMS that store all changes applied to the database, or in the form of a *Version Control System* (VCS).
In any domain example of Event Sourcing we could distinguish two concepts. One is a *command* that indicates requests to the domain for changes. A command might be accepted or rejected. With contrast to the second concept – a command is only a request that might be refused typically in the form of exception. This second concept is an *event*, that is *a statement of a fact that some operation certainly happened in the past within the domain* [10] . Using such a definition there is no mismatch between events (as domain concepts) and the domain. Replaying events, results also in a domain state. Thus such a system is defined in the domain-oriented way. This also radically reduces a number of needed model representations as the events already represent the domain model. *Events* and *commands* are both data structures containing simply data without any behaviour (a.k.a. *Data Transfer Objects* - DTOs). The main discrimination between the two is their intent.
The *accepted command* results in zero or more *events* being emitted to put new facts into the system. One additional concept is an *aggregate*. An aggregate is a stream of events that are somehow related to each other (e.g. action history for one order). Therefore, aggregate is a single object without any reference to other objects or a graph of objects with a root as a starting point of an aggregate.
Therefore, the event sourcing is all about using an application log as a primary source of data. The pattern is based on the assumption that the application state changes should be all present in the form of a dedicated *event* object that captures all information about each change. Now such an object is stored for evaluation of the application state. Therefore, at the stage of application state update, the event object is used as a source of current state information to change the application current state and as a result to make the application's state updated. Since the application state is stored in memory, using the in-memory data structures, working on in-memory data also brings straightforward performance gains eliminating all disk I/O operations and mappings between the disk and in-memory data structures.

Such solution brings a secure storage for state changes. Moreover, at any time application can withdraw its current state and apply the one loaded from the event log object. Therefore, the application state can be recreated – or as we shall use *aggregated* – at any time. For performance reasons, technically it can be realized in the form of application state memory image snapshot that can later instantly replace the current one. Now the snapshot memory image acts as a time break point from which we can recreate all changes in memory owing to the event object change log, thus the snapshot does not need to be made on every change bases. Now the policy of event log start point can be also limited to only log events when the last snapshot is taken.

The event sourcing pattern enables the system to transmit events to multiple systems that can therefore create different states depending on requirements. Due to the fact that the event log can aggregate the arbitrary data model, one can use it to broadcast events for nodes with potentially different models and even schemas.
Event sourcing can obviously also help to store historical information since one can access any state from the event queue log. Up until now, it has been mainly possible by using the data warehousing solutions. Due to such possibilities, the event sourcing also brings the possibility of alternative scenario analysis.

[10] Events are very domain-focused states, e.g. in the general selling case an event might be: ItemOrdered, ItemCancelled, ItemPayed, ItemPreparedForShipement, TransactionClosedWithSuccess etc. Whereas the commands describe rather user domain requests, e.g. PlaceOrder, CancelOrder, CloseOrder etc.

Additional advantages of event sourcing are loose coupling of the current state and the event log state. Moreover, due to the *append-only* manner of events, it is easy to scale such solution. One could also aggregate different data models and schemas based on domain events stored in an event log.

> “As the events represent every action the system has undertaken, any possible model describing the system can be built from the events.”
>
> – *Event Sourcing documentation on Model (see [154])*

The problem with event sourcing is the burden made by the necessity to assure that every system change is captured and stored as event in log. The obvious consequence of event sourcing is that all of the event data must be stored in-memory, and be ready to recover fast enough from the system crash by either re-executing events from the event log or by starting a duplicate system. Additionally, potential event broadcasts can bring side effects when applied to external nodes. The event sourcing might also require additional mechanisms that consider the concurrency access to the event log, or require an event processor single-threaded implementation (just as in the case of event sourcing implementation of *LMAX Buisness Logic Programming* single threaded solution [11]). An enterprise class system should also consider error handling to roll back events in the case the application raises an error or simply to replay all events in a correct sequence. What is more, a bad event is not removed from the event queue, but compensation events are added to provide a true history of the system, which is further beneficial for system audits and traceability.

In the systems such as the one proposed in this dissertation, which is read-based in the case of multiple read requests the postulate would be to base the solution on multiple, read nodes. As an heterogeneous solution it should also consider nodes with different schemas.

3.2.4 Command Query Responsibility Separation (CQRS) Pattern

In the most common approach, an interaction with an information system in the form of application is focused on *Create*, *Read*, *Update*, or *Delete* (CRUD) records from the database. In the simplest case the goal is to store and retrieve data from such CRUD-based data store. However, as the use case scenario moves to some more challenging and sophisticated forms, this approach becomes insufficient. For example, we can easily think of some data retrieval scenario that must be based on multiple sources or even on virtual instances, while additionally considering the information validation. In such cases a designer would have one straightforward option i.e. to focus on the exact domain for data modelling so that the store model becomes as close as possible to the domain model. This is what we consider the *Domain Driven Design* (DDD). The evolution has also brought the layered design that has introduced vertical order for data representation down from the data source, up to the end user that receives a conceptually integrated view resulting from multiple lower or adjacent layers.

3.2.4.1 Command-Query Separation

An interesting approach has also been proposed from a different conceptual level. This new approach has involved some observation that each method is either a *command* that performs action or a *query* that returns data. At start it seems pretty much as a programming language design level particularity, which is actually where it was originally first postulated. It was Bertrand Meyer, who has proposed such discrimination as a part of his work on the Eiffel programming language in [155]. With reference to the layer model, Mayer discusses the *Separation of Concerns*

[11] *Multilateral Trading Facility* (MTF) for *Foreign eXchange* FX market trading; that can handle 6 million orders per second on a single thread.

[12] (SoC) pattern that refers to each layer as assumed to be *correct* (see definition 3.3) conditioned only on the correctness of underlying layers, and thus achieves concentration at separate stages *"on a limited set of problems"* [155, p. 4].

DEFINITION 3.3: *Correctness* AS AN EXTERNAL QUALITY FACTOR AND CENTRAL TASK OF OBJECT-ORIENTED SOFTWARE CONSTRUCTION - ACCORDING TO [155](P.4)

Correctness is the ability of software products to perform their exact tasks, as defined by their specification.

He also claims that SoC *"is essential for maintaining the simplicity of software elements"* [155, p. 363]. The Soc pattern [13] itself, is based on separating program code into distinct conceptual sections, while each section addresses a separate *concern* [14] . A code that embodies the SoC is referred to as a *modular program* that uses interfaces to *encapsulate* information within a section of its code. Another incarnation of SoC presence in a program is – as discussed by Mayer – the layered designed with the presentation, business, data access or persistence layers. SoC brings simplification for management and development of software mainly due to the separations that it imposes. Thus if using SoC, one can easily decompose program into many sections for later reuse, reimplementation and extensions regardless of their coupling with the remaining sections. In the procedural languages separation is done by separate procedures/-functions. In the object-oriented programming we can find SoC while defining separate objects. *Aspect-Oriented Programming* (AOP) uses the objects and aspects for this goal, the same way as the *Service-oriented* approach use services to separate concerns. Additionally, in the case of architectural design patterns such as the *Model View Controler* (MVC) or the *Model View Presenter* (MVP), the separation is straight forward in terms of separation of data-access from the model and the data presentation, that are considered as separate concerns.

Mayer has defined the following principle 3.4

DEFINITION 3.4: *Command-Query Separation (CQS) principle*

Functions should not produce abstract side effects.

In other words it states that the only procedures (i.e. commands) are expected to produce side effects in the form of object change, while not returning any results. This is in contrast to *queries* that provide information about objects but do not change therm. For the sake of understanding Mayer defines *abstract side effect* and *concrete side effect* as follows:

DEFINITION 3.5: *Concrete side effect*

A function produces a concrete side effect if its body contains any of the following:

- An assignment, assignment attempt or creation instruction whose target is an attribute.
- A procedure call.

[12] As introduced by in Dijkstra in [156], and focused around in his work [157]

[13] Realized e.g. by internet protocol stack, HTML/CSS/JS, *Aspect-Oriented Programming* (AOP) (as of e.g. build-in security and logging inside the code).

[14] Which is a set of general (e.g. hardware architecture the code is optimized for) or detailed (class name for instantiation) information that affects the source implementation of the program.

DEFINITION 3.6: *Abstract side effect*

An abstract side effect is a concrete side effect that can change the value of a non-secret [a] query.

[a] The "non-secret queries" (available to specified clients) here – in contrast to "exported queries" (available to all clients) and "secret queries" (available to no client) – means that non-secret query is the one that is exported to selected clients. Now changing the result of the non-secret query is an *abstract side effect* since the change will be visible to at least some clients.

In terms of object-oriented approach, each abstract data type is expressed by the interface. The interface is then offered by class to its clients. The side effect affects the abstract object if it modifies the outcome of any query that is accessible to these clients. However, one must be warned that separation of side effects and return values are not inherently object-oriented.

What is a serious consequence of the CQS principle that will also become crucial for the architectural solution [15] devised in this dissertation is bringing in the referential transparency within.

DEFINITION 3.7: *Referential transparency*

An expression e is referentially transparent if it is possible to exchange any sub-expression with its value without changing the value of e.

The referential transparency is a kind of practical consequence of a mathematical principle of object immutability. For instance $\sqrt{4}$ does not change the number four. An exemplary implementation of such a programming paradigm that would seek to retain this mathematical immutability would be the *functional programming* that lately becomes more present in enterprise development in the form of Scala and other JVM based languages.

The Command Query Responsibility Separation (CQRS) employs CQS by using *Command-Query* objects for data retrieval (`READ`) and modification (`UPDATE`).

3.2.4.2 Command Query Responsibility Separation (CQRS)

CQRS proposes domain model access by splitting its conceptual model into two, separate Command-Query models that can provide less complex model, comparing to common conceptual model for both of these operation types. Such an approach gives great possibilities especially when combined with the command/event based *event sourcing*. One should not forget that such a separation, however, brings questions about keeping the model consistent or eventually consistent. The CQRS obvious profits and advantages can be especially visible while considering complex domains where the separation can bring significant gains.

However, CQRS is not a panacea for all system architectures. In particular one should not consider CQRS as a general system base. It should only be used in this particular subset of system tasks that can effectively benefit from it, and not the entire system. This is of course natural that a unified model of the entire business domain is not a good idea.

> “Total unification of the domain model for a large system will not be feasible or cost-effective.”
>
> *– Eric Evans "Domain-driven design."(see [158, p. 332])*

[15] This characteristic is important in terms of the *remote Data Object Reference* (rDOR) that will be discussed in the next sections.

As a consequence, a system should be divided into separate *bounded contexts* – as it is referred in DDD [158, 159, Respectively: part IV *Strategic Design*, *Chapter 2*] – each of which can then have its own unified, canonical model. Those canonical models can of course overlap while sharing the same structure and can represent only the aspects required for communication. Therefore two bounded contexts might cover unrelated concepts (i.e. complaint policy is only relevant in the complaint support context) and at the same time share some of the concepts (i.e. seller, consumer etc.). While considering integration one can also consider mapping or translating mechanisms between unrelated concepts. While considering the CQRS its postulator claims that:

> “ CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation, a command is any method that mutates state and a query is any method that returns a value). ”
>
> *– Eric Evans (see [160])*

The main benefit of CQRS pattern involves less complex handling of a complex domain. Additionally, one can consider CQRS a right choice for high performance applications. Since CQRS enables simple and independent scaling of the *Command* and *Query* parts of the system, it allows a better system handling – e.g. with many *queries* while a few *writes*. This is due to possible separate optimization techniques that could be applied to both aspects. As for the database integration application we can think of different database access strategies for *queries* (reads) and for *commands* (updates).

CQRS should not, and can not be considered as a silver bullet for every aspect of an application. We can easily find use cases of simple domain applications or those that do not require the *command* part, where CQRS would possibly only deliver overhead and complexity. Another example of application where CQRS would not be the best choice, would be an application with highly coupled request-response communication where the synchronization between the C-Q separate models could additionally provide latency and therefore uncertain gains. Thus, rapid query-response interaction use case scenarios should rather be considered inefficient while using CQRS.

CQRS Implementation Considerations Towards Integration When considering the CQRS to be used for one or more bounded contexts of the integration solution, one of its implementation types can be used. In terms of architecture design presented in section 3.3 the author has considered following CQRS based design types.

Let us assume a scenario when we need to provide an integrated search interface for all employees.

The classical approach would consider the domain model and classes to serve both the *queries* and the *commands*. This approach would consider the `Employee` class and the `EmployeeRepository` repository domain class.

Listing 3.1: Employee class.

```
public class Employee{

  public int Id       { get; private set; }
  public string Name { get; private set; }
  public IReadOnlyList<Task> Tasks { get; private set; }
  public void AddTask(Task task)  {     /* ... */    }
    /* Other methods */
}
```

Listing 3.2: Employee repository class.

```
public class EmployeeRepository{

  public void Save(Employee emp)    { /* ... */ }
  public Employee GetById(int id)  { /* ... */ }
  public IReadOnlyList<Employee> Search(String name)    { /* ... */ }
}
```

While having the `Search` method as a *query* that results in a list of employees, it provides an efficient code without replication nor duplicated functionalities. However, for the purpose of *query* optimization, one would possibly require just a list of employees without the need to retrieve an entire list of their assigned tasks – e.g. simply requesting task count. Querying for each employee's task list can become a significantly time-consuming position on such query evaluation and execution timetable, especially in the case of large numbers of employees, possibly with extensive lists of duties.
Therefore, this kind of design would not fit well a data integration application with read-intense focus.

A more advanced pattern would be to provide additional DTO classes responsible for *queries*, and the remaining domain class responsible for *commands*.

Listing 3.3: Employee class.

```
public class EmployeeDTO{

  public int Id       { get; set; }
  public string Name { get; set; }
  public TaskCount { get; set }
}
```

Therefore, the repository could now modify its search *query* to use the DTO:

Listing 3.4: Employee repository class.

```
public class EmployeeRepository{

  public void Save(Employee emp)    { /* ... */ }
  public Employee GetById(int id)  { /* ... */ }
  public IReadOnlyList<EmployeeDTO> Search(String name)    { /* ... */ }
}
```

While this provides the separation, on the other hand it also brings in the code duplication in domain and the DTO classes. This fails to comply to the good practice of *code reuse* and also the SoC pattern, as the same employee's concern is present in two classes. Additionally, such code

replication breaks the *Don't Repeat Yourself* (DRY) principle of software engineering introduced by Hunt and Thomas in [161, p. 27].

STATEMENT 2: *DRY* PRINCIPLE

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Also E. W. Dijkstra is known for coining the "*two or more, use a for*" rule of thumb relating to the case when processing more than one instance of data structure, it is advised to encapsulate the logic within a loop. On the other hand, there are some less restrictive approaches e.g. approach devised by Fowler in [162] that states that the code can be replicated only once (so called *rule of three*). One could break the DRY principle e.g. for performance reasons like caching to avoid repeating expensive operations but such decision should not be made on a daily and ad hoc purpose basis, such as the discussed design issue.

A more acceptable approach – from the point of view of the query (READ) intense application – would be to provide separated models and thus, APIs for *command* and *query* services. While the remaining domain and DTO classes still provide the code replication, however, now both are part of separate command (domain) and query (read) models and APIs, and thus their existence is better justified.

Listing 3.5: Employee repository class – now handles COMMANDS.

```
public class EmployeeRepository{

  public void Save(Employee emp)    { /* ... */ }
  public Employee GetById(int id)   { /* ... */ }
}
```

Listing 3.6: Extracted query search handler class.

```
public class SearchEmployeeQueryHandler{

    public IReadOnlyList<EmployeeDto> Execute(SearchEmployeeQuery query)
        { /* ... */ }
}
```

The extraction of *query* logic out of the domain model brings optimization towards read intense *query* systems, and at the same time leaves the *command* functionalities intact. This pattern encourages to optimize the *query* dedicated part of an API towards dedicated caching solution or even separate load-balanced server architecture. As already mentioned, such a separation also foster the scalability of the system.

However, the most pure CQRS implementation improves the *query* operations scalability even more. It includes the system separation but also involves the storage separation. Such an approach provides separate data storages optimized for both parts of our system. One can imagine that the operational RDBMS would suit the transactional write queries better than read-friendly key-value store. This can be done with the additional replicated stores. The heterogeneous storage cluster, however, must consider background synchronization and therefore brings only the *eventual consistency*. Therefore the use case scenarios for the NoSQL storage would have to be carefully considered and tailored towards the system expectations and requirements.

The last proposed solution is the one that is the best in terms of scalability for querying operations. However, one must still remember that it comes at the cost of increased complexity and eventual consistency. Choosing the right solution to leverage a particular system should be

faced with the consequences that each approach brings and values against the degree that the target application would have to comply. CQRS itself is not a panacea for all of the system read optimization issues and should be only considered in the bounded context of particular domain. A balance must be found between the degree of separation and the complexity.

In terms of the proposed integration architecture, it is pretty straightforward that the heterogeneous integration problem specification brings encoded requirement for the last and the most complex approach for the CQRS implementation.

3.2.5 OMG CORBA - Standard Specfication

As already mentioned in section Chapter 2.2.1, across many other technologies *Common ORB Architecture* (CORBA) [87] is a serious technical standard, and as an object-oriented, distributed middleware, has become very inspiring for the author. Especially interesting was the way the standard has defined the service integration in the form of *Implementation Repository* (IMR). Despite some other technologies such as Java Remote Method Invocation (RMI), IBM MQ Series, Microsoft's COM and .NET, SOAP, and TIBCO Rendezvous, CORBA remains a general standard that brings for the proposed solution crucial benefits:

- Maturity – developed since 1991
- Open Standard – any closed solution does not provide their detailed specifications; not tied up to a particular vendor
- Efficiency – uses marshalled [16] data for transfer (as a binary buffer rather than programming language types) compared to SOAP using XML with its verbosity it takes much more bandwidth. Additionally, parsing XML is always a CPU intense task. Compared to other solutions CORBA effectiveness looks good, compared e.g. to IBM MQ that requires the user to write low-level marshalling/marshalling code prior to transmission and retrieving of actual (non-binary) data that brings possibly error prone code and introduces additional management complexity.
- Scalability – CORBA can handle large volumes of server data but also high communication load from thousands of applications
- Enterprise tested – wide range of domains where CORBA has proven its point (see [163])

CORBA is basically an object-oriented implementation of older approach of *Remote Procedure Call* (RPC). The *Object Request Broker* (ORB) (or previously RPC) that forms the CORBA acronym, is a method for invoking operations on remote processes – i.e. those running on the same or separate computer. The idea is to make those *remote* calls to look the same as the *local* calls. With such characteristic it might be considered a good choice for a middleware that can play the role of integration software for distinct applications. CORBA is an independent standard since its design can be implemented for distributed nodes working on different architectures and environments, while using most of the high level possible programming languages.

3.2.5.1 CORBA Implementation Repository

A very important feature of CORBA is that it does not separate client from server processes that are being integrated. Each process participating in the CORBA based communication can at the same time – serve objects to remaining present processes, and call for objects from those remaining processes. In short the IMR is a wrapper around database/file that persistently stores information about each of the servers (and their applications) registered with the IMR. Additionally, there is a wrapper that can handle application-to-IMR communication. CORBA provides only partial specification for IMR and thus allows many possibilities for defining how

[16] CORBA terminology for converting data from used programming language types into a binary buffer that can be transmitted

IMR's functionalities can be reached. For example it states only that IMR should know how to access a particular server process and its status.

IMR on Duty for Data Integration Considering all of the CORBA/IMR particularities they can become a really important part for the integration architecture presented in this dissertation. Let us follow the life-cycle of IMR and apply it for data integration. Originally IMR acts as a *central-point-of-truth* about server applications.

The author has discovered an analogy between the location and manipulation algorithms for heterogeneous data that could be put in place of IMR's server applications. The data integration architecture can consider the IMR as a place to store access methods for particular data. Moreover, in the case of carefully designed approach one could also think of supplying it with the data particular characteristics (as described in section 3.3).

Likewise, the data integration solution can be designed to store a modified form of *server/application* information tandem. Such central data integration repository would have to replace the IMR *application* oriented data, by the *data-source/data-details* information. The *server* information from this tandem would then be substituted by the network address details as well as technical contact details of a particular data source. Similarly, the *application* details would be replaced by the data access methods. Storing such information in a central, integrating repository would enable the client data requests to be decomposed and possibly forwarded towards the desired target data in a communication process. This client-to-repository data interchange would have to consider an API provided by the central repository, which could be used for such a communication purpose. As a result of the API call, client would get the target data source contact details including a set of communication characteristics required by the target data source's specific particularities. As a result, client would be able to connect to the adequate data storage compound. Moreover, to reach the exact data from within the data source, a detailed and fast access method, native to the particular target data storage, would have to be embedded within the data provided by the central repository's API. Thus, after settling a connection to the data source pointed by the API call response, client will be able to use the adequate and expected query statement towards the specific data storing engine.

The data-source/data-details contained within the central-integration repository, and made available by its API are actually considered as a metadata – i.e. the information about the data sources and their existent, real data contents.

3.2.6 Metadata

A wide spectrum of metadata utilized by numerous, modern solutions has already been discussed in 2.4.2. However, for the purpose of the discussed solution, a well-tailored metadata representation would have to be provided. A metamodel for data intergation and location must be flexible enough to face all potential issues that might originate from the fragmentation of replication data characteristics.

The most widely spread and accepted standard for metadata, often referred to as *lingua franca* of the integration is the XML. XML is sometimes marketed as the solution to the semantic heterogeneity problem. Nothing could be further from the truth. Just because two people tag a data element as a salary does not mean that the two data elements are comparable (See [164]).

However, the metadata is the key aspect of the presented integration solution. Each data element present in the integrated data sources grid will be represented with the structured metadata that conform an integration or contributory view.

3.2.7 Design Patterns - Study of Utility

Referring to the general assumption of CORBA IMR – used by the author in the development process of the integration architecture – a careful study can bring some interesting but challenging

findings. First of all, considering the CORBA IMR data integration architecture, it is highly probable that a central repository will became a target to multiple concurrent client requests. As a well designed system this should even be able to handle up to ten thousand clients simultaneously [17] . There have been multiple strategies for handling effectively such cases (see [165]). However, the C10K class of problems deals with the extended set of issues focusing also on hardware, operating systems memory management particularities or network stack mechanisms.

Therefore, author has focused only on software design patterns that can be applicable for software architecture for data integration.

3.2.7.1 Reactor pattern

As prototyped by CORBA IMR, also in the discussed integration architecture, communication with client data requests must be handled effectively even in the case of multiple requests. Here is where the *reactor* design pattern comes in. Actually numerous CORBA implementations [166] (TAO [167], VisiBroker, or Orbix) also uses the reactor pattern in the ORB Core [87] layer.The reactor is used for demultiplexing and dispatching ORB requests to servants. The reactor aims to serve the server application for concurrent multi-client request handling.

The reactor is one of the most well known (since 1995 – [168, 169]) design patterns for simultaneous event-handling of service requests [18] . In the case of architectural design, an event (in contrast to domain event from event sourcing) is *an action or an occurrence that happened during program runtime, that must be handled by the program*. Events are typically handled by the dedicated part of the program that dispatches events/messages concurrently in form of the *event loop* [19] .

INFOBOX 6: *Event-driven program*

The application that changes its state and/or behaviour in response to incoming events is called event-driven program.

However, the *reactor* is more specific than the general event-driven approach, and can effectively use *event loop* to block resources [20] . The problem that the reactor solves is to handle multiple, concurrent client service requests. Structurally, the reactor is a synchronous demultiplexer of events, that dispatches them to their adequate event handlers. Each *event handler*, actually processes a certain type of *events*. Additionally, prior to be used, each *event handler* must be registered in an *initiation dispatcher* . The pattern work flow is simple. Multiple, multiplexed events that reaches the *reactor* instance are target to synchronous event *demultiplexer*. *Demultiplexer* then notifies the *initiation dispatcher* about the incoming new *event*. As a result, the *dispatcher* synchronously calls back the *event handler* that is dedicated to such new, particular *event*. Finally, the *event handler* dispatches the event to the *concrete event handler*, i.e. the method that implements the requested service in the application-specific manner.

In the discussed application context this application can be a part of the architecture API for client calls for the integrated data. This will be elaborated in section 3.3.2.3.

---

[17] This is often referred [165] to as *C10k problem*, that has to deal with hardware considerations and multi- or single- threaded model for handling with concurrency. In this case we refer to concurrent requests.

[18] A.k.a *notifier, dispatcher*.

[19] One of the methods for communication between processes and flow control (often main loop of the program). Used for dispatching events from event providers to event handlers. Mostly operates asynchronously. Event loop is a programming structure that continuously checks event sources/providers for new events, and if a new event occurs, event loop calls the respective routines to handle the event.

[20] In the case blocking is not required to start synchronous operation on resource, the resource is sent to the dispatcher.

Reactor Pros & Cons The most intuitive, alternative solution of the problem approached by the reactor pattern is multi-threading. Threading, however, can lead to performance issues [21] , might require complex concurrency control schemes and what is more, cannot be considered on every programming platform. Thus this approach might not be effective, simple to program, nor portable between multiple platforms [22]. The synchronous nature of reactor's request handler calls, enables concurrency without multi-threading complexity. This also brings more portability of reactor-based applications.

The most notable benefit of the reactor pattern is the Separation of Concerns (SoC) – see section 3.2.4.1. The reactor provides decoupling between the reactor implementation and the application code. Thus, one can reuse the application-independent components, while still handling application-specific methods for event handling.
Additionally, since *handlers* force the functionality decoupling with separate classes [23] , Reactor also improves modularity and thus reusability. The reactor pattern, on the other hand, is hard to debug, since the inverted flow of control involves reactor framework and the application method callbacks. This bring some additional complexity to step-by-step debug of the application as its programmer might not have access or comply with the framework code.
Additionally, while using *reactor*, one must be aware that as a single-threaded application, its *event handlers* are not preemptive (i.e. tasks can not be temporarily interrupted without their cooperation, with the intent of resuming at later time) during execution, as blocking one handle can stop the entire process. In other words, the context switching is not possible.

3.2.7.2 Reactor Related Patterns Considered Applicable for Client Interfacing

The non-preemptiveness of reactor is crucial in the cases when long-duration operations are involved, like large image (e.g. medical [171]) processing. In such cases one should rather consider patterns such as Active Object [172], that uses multi-threading instead of single thread to complete its tasks. It simplifies synchronization of shared resource access by invoking methods in different threads of controls, thus decoupling the method invocation from the method execution.

The *Reactor* pattern itself is often considered related to the *Proactor* pattern [173], which in contrast to the reactor is asynchronous. It simply demultiplexes and dispatches the event handlers triggered by the completion of asynchronous events, while the reactor triggers event handlers when it is possible to initiate synchronous operation without blocking.

Referring to more general patterns as of [1], there are two that show some resemblance – i.e. the *Observer* and *Chain of Responsibility*(CoR) patterns. The Observer aims at handling a single source of events, all actions are commenced on multiple dependent instances when the single source induces. In the latter, CoR pattern, client request is forwarded to the responsible service provider, that is the first matching Event Handler in the chain. In contrast, the *reactor* matches the concrete Event Handler with a specific source of events.

3.2.8 Integration Database Model - IDBM

While the general way of interfacing can be approached by the reactor based model, there is still a considerable amount of issues regarding the central integration repository model. As already mentioned in section 3.2.6 metadata will be playing a significant role as a data description language. Regarding the specific nature of virtual and metadata-based information stored in the central repository one would have to consider a dedicated datamodel. This is due to the fact that

[21] Such as context switching, synchronization and data movement (see [170])

[22] The solution to raw threading could be with the use of frameworks such as Akka that use the actor model abstraction over threading complexity, while also providing scalable and real-time interfaces for implementing concurrent and distributed applications.

[23] *Single Responsibility* principle from SOLID acronym, mnemonic for OO design rules.

the existing data models like the row-, column-, etc., are designed to handle a particular data and to face its specific characteristic. While this is the right approach per se, when we know the target datamodel, however this tends to become problematic and unclear, if the actual data that is going to be represented and integrated is unknown. Moreover, queries towards the datamodel will be ad-hoc, and mostly unexpected due to a lack of submitter constraints. All this issues deserve to be considered in a dedicated way.

Therefore, in the following sections the design decisions – of proposed solution – will be introduced and justified.

General Data Model Design To achieve the quest of providing integrated data transparently it is not enough to have only a metadata scheme. A conceptual datamodel is required to present replication and fragmentation particularities. Moreover, as we deal with a heterogeneous and distributed environment, it is also expected to represent the integrated data views transparently. The proposed solution is to introduce a new, *concept-oriented model* (COM) for global, integrated schema. The *concept* here, is referred to as a basic, descriptive component of the integration schema view that represents distributed data particularities [24] . This new model, physically represented by metadata, assumes that each unique data value must be retrieved in an unambiguous way, despite its replications and fragmentations. To assure such explicitness, one can easily imagine that a key-value model will be essential in such case, to maintain context of each value. Additionally, single-point-of-reference for every stored piece of data will provide a great advantage of saving physical space, compared to e.g. insufficiently normalized relational databases, and most of the non-relational ones. Since each raw data *concept* is present only once, in the *concept* oriented model, its normalization is by-design. Simple matedata value-to-source mapping function however, is not enough. The model must also consider a *concept* for referencing data replications or relationships. The reference type value *concept* should be stored the same way as the regular raw data concepts, and thus enabling transparent navigation.

With such assumptions, an index-based metadata storage with the referencing conceptual entity would be enough to represent and obtain each part of the requested record together with its replicas and possible fragmented parts regardless of their localization.

Organization and Purpose of Replicas Replication nature, in a distributed and heterogeneous data grid, can be intentional – to assure high availability and/or disaster recovery, or might originate in legacy nature of integrated data sources. Therefore, representing replication in the dedicated concept-oriented data model, would benefit from a common, unified model that could provide enough abstraction to represent any other datamodel. This way one could intentionally replicate and store data in the datamodel that would suit it the best regarding its purpose. For instance, a transactional query would be sent to the operational replica that has the relational origin, while for scan queries (that focus more on aggregations and summaries) one could use column store replica etc. This way individual client data requests might be mapped to the dedicated queries depending on query nature. The proof of a testing scenario is presented in the final chapter. As for the legacy replications that originate in past data migrations, the integration architecture can use them simply as a recovery source or a load-balancing target.

3.2.9 Indexing Role in Integrated Datamodel

As already mentioned, the datamodel proposed for the integrating architecture is based on already well accustomed key-value paradigm – for accessing data unambiguously. This paradigm is already present in the relational (primary key), object (object id) or NoSQL (MongoDB:ObjectId _id field, Cassandra: composite/simple key; Neo4j: id_property etc.) data stores. In the integration architecture the ordered *key* values are responsible for unambiguous locating of the

[24] E.g. attribute, tuple, record, fragmentation component, replica, etc.

contents of each schema *concept* (e.g. record, fragmentation component, replica, etc.) of data from the integrated grid.

Since, in the proposed architecture, the *key* values for each schema conceptual component are auto-generated (incremented) at the stage of creating the integration/global schema, the schema (on concept level) becomes unambiguous and complete while preserving the context of each concept. This nature makes the *key* values to act more as a *forward index* or, in some optimization cases, an *inverted index*.

Forward Index As the amount of data stored in a database has significant impact on its performance we should note that, the bigger data size is, the more the indexing is important to the system. In a proposed solution, one has to be aware that it is potentially designed for handling accumulated BigData sizes from multiple sources in a complex schema. Since the architecture is targeted for data integration, the amount of data made available will become a sum of all integrated data volumes.

The problem with persistent data access is that (even when dealing with metadata) it is grouped and stored on disk in blocks of data. To access data from within a block one needs to access the entire block in an atomic manner. Each block is organized like a linked list (i.e. it contains data section and a section of pointer to the next block). Thus continuous data access is imposed. Assuming that one can sort records based on one attribute, finding the requested record in linear search, for N blocks (that the table spans), would require $N/2$ block access. Moreover, if the wanted attribute is not unique the number of search block accesses becomes N. Now a simple sort on the requested attribute can profit from binary search of $log_2 N$ block accesses. Additionally, if the data is sorted, all searched value instances can be found by simply moving forward the sort order until the first different attribute value found. Indexing, as a sorting method, brings the advantage of having sorted access based on record attribute values. Therefore the performance increase is substantial. One main issue of index, is that it takes a physical storage space.

However, in the proposed solution of the integration architecture, the indexing based model will be targeted on usage of structured metadata stored at a dedicated storage, thus making this downside relatively harmless.

Indexing will also become an important optimization technique for searching the distributed heterogeneous data that has been integrated by the discussed architecture. Please refer to section 4.2.1 for details.

In the integrated environment with the complex metadata, the speed up of the searching based on the indexed attribute, makes indexes application desirable for the integration data model. This is why the key-value approach determines how each single part of the integration schema will be covered (see section 3.3 and Appendix A for details).

Now the unique *key* values in the *key-value* approach of the presented architecture play the role of a *best record id*, and at the same time – the values of the table forward index. This is due to the fact that each *concept* is going to have its unambiguous key, but what is more, all of the concepts (attribute / cell, tuple, record) that belong to the same record must share its key – i.e. the *best record ID* (BRI) playing the role of forward index.

For instance, let us assume that one record attribute is stored in three different replicas, that additionally involve different fragmentation patterns (Figure 3.4). Now, trying to reach for all replicas would require reaching for the *key* index value of the attribute record, to retrieve the first replica, but additionally, also it would be enough to investigate the attribute metadata, and to access remaining replicas *concepts*, due to attribute embedded reference type. This would enable direct access to selective fragmentation pattern contents of potentially vertically fragmented incomplete (i.e. missing some attributes) records. Additionally, it eliminates the need of storing the *key* reference for each store that holds only tuple (with the requested attribute) instead of

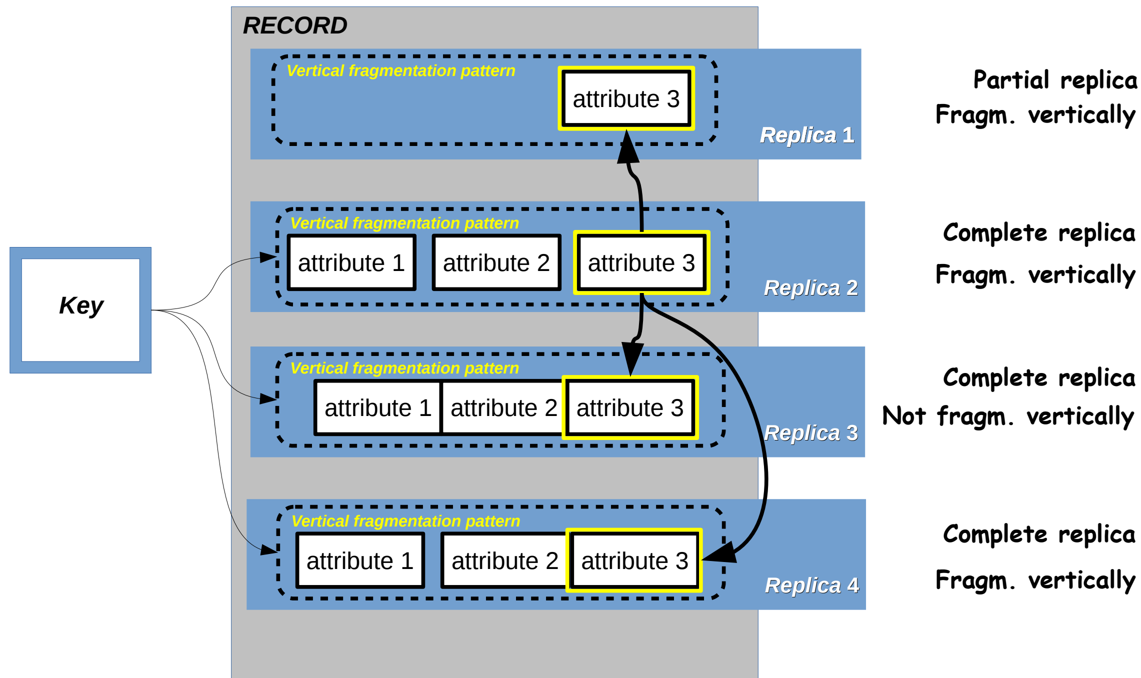


Figure 3.4: *Key*, as an index, allows access only to complete record representation. Accessing single attribute/tuple replicas only from within the complete record replicas.

complete record. However, it imposes one condition. For each record *key*, there must be at least one complete record representation regarding vertical fragmentation patterns. See Figure 3.4.

Additionally, due to separate *key* identification policy for each conceptual domain – i.e. the keys for entity, record, attribute, tuple, replica or fragmentation component – it also becomes simple and efficient to distinguish between the nature of the *concept* referred by the *key*.

3.3 The Architecture

There are two compelling reasons for using the CQRS pattern for the proposed architecture. First of all, an enterprise-class application in modern software reality is mostly composed of multiple layers like: UI code, REST / SOAP service for client calls, transformations from DTOs, validation, business logic, repositories, data access, DAOs etc. While this is of course good for its layered model and required by the *command* part of system interactions, it might be considered superfluous for the *query* operations. Therefore, in terms of integration architecture based on read access, it is most reasonable to provide some separations that will involve the CQRS concept within particular bounding contexts.

Additionally, what is important is that for the auditing sake, it would also be desired to consider equipping the CQRS with an Event Sourcing mechanism.

3.3.1 *Principia* – Assumptions and Directions

As described in Chapter 2 integration has many faces and flavours. Therefore, to expand the topic architecture the author has provided some rules of thumb and assumptions. The presented approach faces the following integration challenges:

- Providing a central point of reference for transparent access to the data persisted at many sources.

- The integrated data sources can potentially be distributed.

- Each data integrated from the data source is expected to be served by different storage engine (database provider).
- Data at the integrated sources can have different data schemas from each other and from the single-point-of-reference integration / global schema
- Each data source can use an arbitrary data model for local data (i.e. relational, object-relational, object, nosql etc.)

The proposed architecture has also made some integration assumptions and decisions:

- Use a well known, structural model of communication (i.e. XML, JSON, etc.)
- Used technologies must be simple and lightweight.
- Provide client-server peer communication rather than central integration hub (less complex central computations, target data does not play any part in the integration process)
- Use main, central metadata repository as a single-point-of-reference for the integrated data.
- Use the central metadata integration repository only for the data "*addressing*" purposes.

Despite its flexibility and versatility, the generic proposal of the architecture is not aimed for distributed processing, nor data mining. Namely this dissertation considers out-of-scope issues, such as:

- transactions or distributed transactions,
- SQL query processing and optimizations,
- inter-table relationships (only one virtual entity is considered),
- data source vendor / model dependent mechanisms,
- contributory to global (integration) schema strict mapping rules

The above aspects, while not elaborated, however, can become a target for future research [25] . In general the architecture works in a *Data-Store-as-a-Service* (DSaaS) model that treats each of the integrated data sources as a service. Thus the goals of the solution and main targets take their origin from this model. Those can be presented as in the following list.

- Do not interfere with the legacy data source storing / querying methods and optimizations, as it tends to be best suited for its particular nature already by its vendor.
- Legacy data source is stable, well known and accustomed by its local users, and does not require new knowledge learning.
- One service interface should poses all information about physical storage particularities of the target data
- Aim at data integration, not the database integration.
- Integrate not all available data, meaning integrate only those data that are required by the general integration schema. That is regardless of the local schema, data model vendor, etc.
- Use native queries while reaching the target data at local data stores.

[25] In Chapter 4 one can find one example of query optimization technique with the use of the proposed architecture.

***Bounded Context* Based Caononical Messaging Model for Data Integration and Unification.** Most of the industry solutions for the enterprise class data modelling tend to create one common and ubiquitous conceptual model. To express the fundamental requirement for integration model one must assure that the model must be internally consistent in terms of *unification* (see Definition 3.8).

DEFINITION 3.8: *Unification*

The model internal consistency is referred to as *unification* if the model contains concepts with the same meaning across the domain, and there are no contradictory rules

As a consequence the model is meaningless unless it is logically consistent. The straightforward consequence might be to create one common domain model that would cover all aspects. This, however, tends to be only an idealistic solution, since the integration of heterogeneous systems is too complex to maintain such a monolithic, and complicated model.

An alternative approach is represented by the presented solution. Here each data source would have its context suited model residing at each data source side. Of course, each context must be structured and unified while bounded to the specific data source environment. Sometimes different contexts can cover the same parts of the domain in a different way that is typical of local data source, but this is irrelevant at the level of each data source. In the case of the proposed architecture, the only factor for drawing boundaries between different data source bounding contexts would be purely based on their heterogeneous nature. This is due to the usage of messaging, that enables canonical integration at the level of mediation across all data source nodes. These contexts in terms of DDD are referred to as a *Bounded Context*s.

Moreover, as for the *Domain Driven Design* (DDD) it is claimed that:

> "We need ways of keeping crucial parts of the model tightly unified. None of this happens by itself or through good intentions. It only happens through conscious design decisions and institution of specific processes.
> *Total unification of the domain model for a large system will not be feasible or cost-effective.*"
>
> *– Eric Evans on Maintaining Model Integrity in Strategic Design (see [158, p. 235])*

As a consequence, in the proposed solution the design establishes possibility for multiple models on multiple heterogeneous sites to reside, while integrated at the level of higher, integration layers. This is also an obvious advantage of the proposed messaging-based architecture over other integration solutions based on common or shared database approach. The messaging enables to forget about areas that would have to be unified – between the central repository and local data sources – in other case. Therefore, one could use:

- couple canonical models instead of forcing one at the integrated sides / nodes
- models that overlap (describe the same terms from the integration-domain semantic area with the data source particular approach [26]) at the stage of integration
- many models with translation at the mediating abstraction layer where model overlapping might occur across multiple data sources
- models that cover only selected parts of data characteristics fundamental to integration
- live data source registering procedures rather than static connections made in advance

[26] For instance, the unique record id at local data source in the relational model would mean the *primary key*, but in the object model it would be the *object id* or a *key* value in the key-value store, etc. In that way the author refers to those terms as *polysemic*.

- simple canonical model for hub based n connections instead of p2p n^2 connection burden

With such approach the integration architecture is broken down into smaller modelling subclasses that can be simpler to manage and adopt.

⋆ ⋆ ⋆

Most of the modern state of the art solutions that approach this class of problems require homogeneous integrated environment from the vendor's point of view or at least the same data schema on all integrated data sources. Even if there is a set of tools that can overcome these limitations they tend to be strictly vendor-oriented or closed-source, thus obfuscated and undisclosed for the end user for understanding or extension. In contrast to this existing solution, the advantage of the proposed solution is its maximized flexibility, extension-openness and independence.

3.3.2 Components of the Architecture

For the analytical purpose one can accept the top-down approach as the one to make the design decisions. However, as for the implementation part, the *bottom-up* strategy tends to be less error prone, as some detailed issues that have not been foreseen at the design stage may arise. In such case, the *top-down* approach would require additional iteration of already created upper part of the design, generating efficiency drop.

In the following sections, the author has described the solution components in a bottom-up fashion.

3.3.2.1 Mediation Layer

While considering the architectural design in a layered model the data access layer would be the bottom-most one. Its goal is to provide an interface at each data source site, for communication with the rest of the integrated environment. It also provides local *bounded context* translation for higher, integration layers in the form of *mediator*.

The general principle for each data source, prior to be registered within the integration central-point-of-reference, is to conform to the common communication protocol. This protocol would be unified across the integrated environment. The system architecture component that is going to provide such interface and service at the legacy data source site, would be the *mediator*. The mediator as a component of the architecture has a significant role in communication. It is used for connecting the data source to the main metadata repository, registers its available data and provides access methods for local data to the central global integration schema. The mediator also translates between the local bounded context and the global messaging scheme.

Since the implementation of each *mediator* itself is component based, it consists of the mediator communication component, local data source access *wrapper* and the *adapter*. In the proposed approach, the *adapter* is responsible for covering the particular local source characteristics. It would have to cover the data access layer techniques. For instance, in the context of relational model, the object-relational mapping could take part in connecting to the data source. In the area of NoSQL this could be as well, the low level JDBC driver. While the nature of the data is moved to the programming language model by adapter [27] the *wrapper* would have to provide the business logic. It would have to involve the set of operations that would have to be enabled on the local data. Thus, on one hand, *wrapper* would have to be aware of the adapter functionalities, but on the other hand, it would have to serve as an interface [28] for the *mediator* requests.

[27] In the case of relational paradigm this will cover the impedance-mismatch. In NoSQL it would simply focus on providing the raw *data transfer objects* (DTOs).

[28] More in a data access object (DAO) fashion.

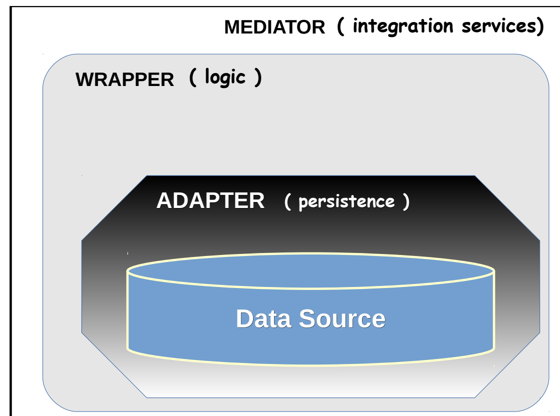


Figure 3.5: Mediation component layer model.

Finally, the *mediator* – as a top-most local layer – would have to provide service interface to the underlying local data already covered by wrapper and adapter. The services would have to cover the approved general communication scheme in the integrated grid.

The structural character of the metadata, describing each of the stored entity particularity, fits well for storing the metadata from multiple data sources in a document store such as e.g. MongoDB. To justify the usage of the document store, for storing a complete set of metadata about the local data – or as it will be referred to *contributory views* – one has to be aware of what kind of information is going to be covered with it. The contribution of each integrated data source, is formulated by its dedicated *mediator*. The output of the mediator is a raw, metadata description of what is made available for access from within the integration architecture. Thus, such metadata view of each legacy data source being integrated must cover the local resource's data schema. On the other hand, the model of the metadata must be elastic enough to represent universal schema that can later become part of the global *integration view* unambiguously, regardless of what data source model it originates from. The author proposes to use a well known semantics based on the relational terms of *contributory schema* and *contributory table*, or shortly `C_SCHEMA` and `C_TABLE` respectively. In such case, for each covered model, there is a need for semantic interpretation of what will the `C_SCHEMA/C_TABLE` tandem mean in the case of models other than relational.
For example, for the sources accessed with JDBC driver the `C_SCHEMA` would become a database schema as defined by JDBC, while the `C_TABLE` would represent any tabular structures such as table, view, etc. In the case of integrating some NoSQL solutions like MongoDB, the simple mapping would be the MongoDB `database` – as a physical container for `collections` – to be the `C_SCHEMA`, while the `collection` would be mapped as the `C_TABLE`. Despite the fact that in MongoDB, one collection holds documents that are not obliged to enforce a schema and thus, can have different fields, typically all documents in a collection have a similar or related purpose according to the convention of `namespace` [29] .

For the purpose of multi-model contributory view schema, an extended semantic mapping has been proposed and accustomed – see Section A.1 for details and examples.

Additionally, the raw schema metadata is not the only thing that is being passed from mediators as a *contributory view*. *Contributory view* also contains detailed retrieval information for each particular `C_SCHEMA/C_TABLE` contained within the metadata. This information would be represented in the form of native, *Fast Access Methods* (FAMs). FAM will refer to a model specific, native query that can retrieve every single schema element based on the fastest-possible-way available at the certain data source. For instance, for the relational model FAM would be a SQL *selection* query based on the *primary key* values.

[29] I.e. `databaseA.collectionA.collectionB` would be considered as `databaseA` as a `C_SCHEMA` and the `collectionA.collectionB` as a `C_TABLE`.

Listing 3.7: SQL based FAM selection

```
1 SELECT *
2   FROM   products
3   WHERE  pk BETWEEN 1 AND 1000;
```

This is due to using index in the form of *primary key* in the relational model which is the fastest possible way of reaching the data represented in the relational model. Since the functionality provided by *primary key* in the relational model in other models differ e.g. *object id* in the object model, *_id field*, *row key* for Cassandra, *id_property* for Neo4j etc., in the discussed architecture this concept will be commonly referred to as a *best record id* (BRI).

The choice of the document model is a consequence of a postulate from Chapter 2 (Statement 1) to choose the data model based on the target data nature. Here, the document store model reflects and expresses the features of structured nature, that are considered crucial to represent the target data, which in this case is a deeply structured metadata. Additionally, within the metadata, the mediators embed the local native access methods used later by client for target data retrieval.

Metamodel of *Contributory View* Each integrated resource would have to get its own mediator that fit its data model. The mediator would have to be configured to connect and access the data source. Therefore, each mediator prior to initialization needs to get configuration details for particular data source – namely the connection details, user / password, target database, load metrics etc. Then each mediator initiates metadata collecting procedure, that is data source specific. For example, in the case of relational model, one can use one of the ORM solutions – such as Hibernate – to investigate the database schema and to intercept native queries that would be the FAMs used for getting each table content. As a result the contributory view would have conform the schema represented in the form of regular (non-optimized) connection details structure in Listing 3.8.

Listing 3.8: *Contributory View* metadata schema. Some parts omitted for readability

```
1  REGULARCONNECTIONDETAIL {
2          [...]
3
4          typedef enum NATIVEFASTACCESSMETHOD{
5                  PK, OID, _ID, CompositeKey[, ... ]
6          };
7          NATIVEBRI{
8                  NativeFastAccessMethod nFAM_f;// flag for fam ie PK for
                       RDBMS or OID for ODB
9                  sequence<string>                bri;
10         };
11
12         FAM {
13                 sequence<NativeBRI>      nBRI;
14                 string  accessMethod;    // query for RDBMS
15         }:
16
17         OBJECTBODY {
18                 [...]
19                 FAM                accessMethod;
20                 sequence<string>                attributes;
21         };
22
23         DETAILS{
24                 string                  host;
```

```
25                  unsigned short          port;
26                  CommunicationConf       protocolSpec;
27                  ObjectBody              object;
28                  [...]
29          };
30  };
```

Due to already mentioned similarities (discussed in section 3.2.5.1) between the topical architecture and CORBA IMR concept, all of the listings description will use the IDL-like [30] language, which is also used by OMG to describe the CORBA standard.

The connection details – apart from some host/port networking data addressing, protocol specification and some other communication configuration options, discussed later – contain the `ObjectBody` that is responsible for storing detailed information about each conceptual element of a contributory view metamodel. As already mentioned the architecture will be based on *concept-oriented model* (COM). The concepts of entity, record, tuple, attribute etc., will be expressed using this model right from the lowest layer of legacy data source data representation. Therefore each concept of legacy data is described in detail, using the `ObjectBody` structure. When referring to the concept of entity, the `ObjectBody` field contains a list of its attributes represented by a sequence of strings, and a FAM. FAM is described here with a string representation of native query, ready to be committed towards the data source to retrieve the described entity, and the `BRI` sequence of best record identifiers for the selected records out of the requested table. Additionally, just to identify what kind of BRI is served according to the legacy data source the `NativeFastAccessMethod` enumerates type used for embeding such information.

Even though Listing 3.8 does not contain complete functionality required by the architecture, it clarifies how the requested legacy data model concepts can be represented, and how the contributory view will store its description in the unified across the integration architecture form.

The connection details information is formatted as described in the listing, and is further delivered to the central-point-of-reference, integration layer by mediator in the form of contributory view.

3.3.2.2 Integration Layer

The mediation layer role is to bring data interactions to a higher logical level where the local data source particularities do not have to be taken under consideration. The integration layer goal however, is to provide some canonical messaging infrastructure for virtual *Integration Views* [31] , and a central-point-of-reference for the client data requests. The integration layer is able to communicate with all of the registered mediators in the integrated grid. In general it has three major tasks to complete.

- Collect and gather complete metadata about each of the mediator-registered data sources, and store it in the form of *Contributory Views* – one per integrated data source
- Enable to build *Integration View* (a.k.a global view) out of the picked, arbitrary *Contributory Views.* [32]
- Provide the possibility to apply transparent access optimization techniques [33] for *Integration Views.*

[30] Some syntactical parts such as `struct` or `module` keywords have been omitted in the Listing 3.8 for the sake of readability. One can refer to fully IDL compatible version in Appendix A.1.1

[31] Which will be discussed in detail in section 3.3.3.

[32] Integration Views represent the abstract schema that is available for client data requests.

[33] See Chapter 4 for three examples of optimizations.

Thus, the basic functionality of the layer is to integrate and prepare the dedicated Integration View(s) that later will become indirectly (through *Interface Layer*) subjected by the client requests towards the integration architecture. Since the integration layer has to store legacy, integrated data source schemas (i.e. Contributory View), and virtual arbitrary global schema (i.e. Integration View) to make them available for client calls, both real and virtual schemas might be represented with different persistence approach. The target virtual, *integration view* details will be elaborated more in the following *Qboid* paragraph. As for the *contributory views*, in the simplest case, it would be stored in a document based model (see section 2.3.5.2) – e.g. MongoDB instance – due to its structural nature.

***Integration View* – Challenging the Distribution Issues with the Qboid** Since the document model and its implementation in the form of document store proved to be suitable for handling the *Contributory Views*, it is straightforward that the generic implementation – for ease of cooperation – can employ the same model which will be well suited for the *Integration View*. This is due to the fact that the *Integration View* stores a virtual, global schema that is primarily based on the metadata originating from the *Contributory Views*. Despite the fact that the document model tends to be a simple and clear solution for the *contributory view*, the *integration view* has to implement and express a more complex schema, and thus requires a more extensive approach. The complexity of *Integration View* originates in need for representing all distribution particularities of the virtually integrated target data. To face such challenge, the author has devises a solution based on a dedicated design for data representation. This design is focused around the concept of a *Qboid*.

The Qboid role is to provide:

- a fastener, for faster joining integrated data sources to the global data integration view,
- platform for virtually integrating data from the underlying resources in a common manner,
- and the design concepts, and mechanisms that can handle the distribution issues and particularities.

As an intuition of Qboid is responsible for covering the distribution issues such as fragmentation patterns and replications, the author has discovered that those three main aspects could be easily represented in the form of a conceptual 3D cuboidal shape. This form will become scaffolded by organizing the fragmentation patterns and the replication in three dimensions – similarly to how the OLAP cube is built out of data dimensions.
When we place vertical and horizontal fragmentations and replication characteristics as a three perpendicular axis we can get the 3D frame of reference that will be used as a scaffold for existing combinations of those three aspects of integrated data. This representation of Qboid is entity-oriented (`C_TABLE`), meaning that the Qboid is essentially an emanation of data that belongs to one virtual *integration view's* entity, based on selected `C_TABLE`s. As described in Figure 3.6, each part of the Qboid schema that represents a separate part of fragmentation or a replication pattern, is represented and marked by the separate interoperable *Database Object Reference* (DOR) object. The role of each DOR, is to contain all important `ContactDetails` regarding the data that it represents. All data particularities, network location, connection methods, local data privileges and different contact profiles are included within.
This way in the *concept-oriented model* (COM) each concept becomes represented by a multiple DORs that would cover all particularities.
Along the `Concept` module, the most basic structure would be the `AccessObject` structure.

Listing 3.9: Remote Database Object Reference (rDOR)

```
1 module CONCEPT{
2
3         typedef enum INTEGRATIONVIEWCONTEXT{
```

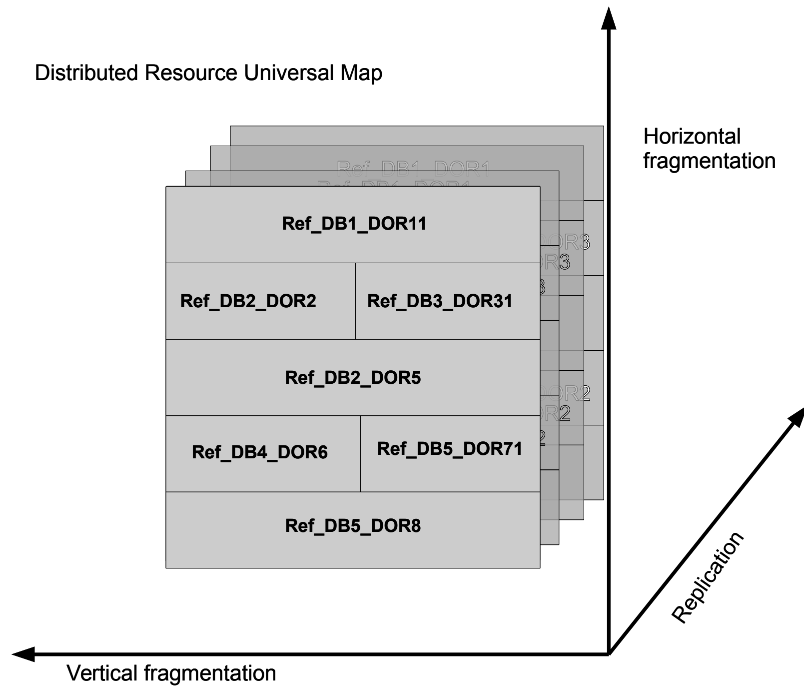


Figure 3.6: DRUM – Database Resource Universal Map

```
4                     ENTITY, RECORD, TUPLE, ATTRIBUTE[,...]
5           };
6           typedef unsigned long GlobalBRI;
7
8           struct AccessObject {
9                     string                        repo_ID;
10                    IntegrationViewContext        iv_Ctx;
11                    sequence<GlobalBRI>     gBRI;   // only for complete ENTITY
                          or RECORD
12                    sequence<Qboid::rDOR>    profiles;
13                    sequence<AccessObject> iv_Replicas;
14          };
15
16          struct IntegreationView {
17                    string                  iv_ID;
18                    sequence<AccessObject>  concept_AccessObject;
19          };
20  };
21
22  module Qboid {
23
24          typedef enum SourceID {
25                    SELF, POSTGRES, MS_SQL, MYSQL, ORACLE_11G, MONGO_DB
26          };
27          typedef unsigned long DOR_Id;
28
29          struct rDOR {
30                    DOR_Id                  dorID;
31                    boolean                 vert_Fragm;
32                    SourceID                src_ID;
33                    sequence<ContactDetails::ConnectionProfile> object_Refs;
34          };
```

```
35 };
```

The *access object* would be storing the `repo_id` field which would contain the data source repository id of the data described by particular remote DOR (rDOR). The repo_id value would also be used to express the vertical fragmentation of the described data in the global integration context [34] . The `iv_Ctx` enum, self explanatory values describes how should the target data be considered in terms of the integration perspective. The third field is a sequence of `rDORs`. Each *rDOR*, here refers to the nature of the DOR-described data storage method at the data source, and how it has been configured for access. The last field of the access object stores reference to potential replicas of the current access object.

As for the Qboid, it is mainly focused on covering the distributed and local data particularities. Its main content is the rDOR that covers type of the data source, network profiling or integrity check rules as well as potential vertical fragmentation and native fast access methods (FAMs). For detailed example see Appendix A.1.1.

Two Contexts of Integration The contact details provide an entire spectrum of access and optimization methods for each *concept* represented by DORs. However, to identify a *concept* (or a DOR), which is unique within the *Integration View* (or the Qboid process), a dedicated identification mechanism must be supported. This is due to the fact that along the integration process we deal with two *bounded contexts.*
The first one is the context of *Integration View* focused around the virtual integration schema. It covers each entity (table) concept and its content, in the form of collection of BRI-based record conceptual representations. The *Integration View* context uses DORs to cover all issues caused by the fragmentation and / or replication characteristics of each concept. Thus each *concept* reference to a set of DORs that are thought to represent this concept. This way the *concept* becomes the context's basic work unit and is responsible for having a unique identity that is unrelated towards the contained DOR's ids. In other words, DORs in *Integration View* context are building blocks for each concept.
The second is the Qboid technical context that is focused on representing all data that is to be integrated within the architecture. Thus each piece of data is represented by a single DOR. However, in the context of Qboid – where each DOR object can contain multiple DOR sub-objects [35] – a set of DORs (even in the case of describing the same *Integration View* concept) must be discriminated between each other with its own DOR id. This is due to the fact that the *Integration View* context is based on the concepts of tables, records, replicas, tuples etc., all constructed out of DOR objects (see Figure 3.7) and identified by the BRI. Therefore, while considering the Qboid context one has to deal with a set of DORs that are referenced between each other, while no BRI equivalent. In other words, despite the fact that particular BRI can refer to the DOR, the BRI is not a discriminating factor in the context of Qboid as many DORs can be referenced by the same BRI. Thus to obtain the goal of DOR unique identification in the context of Qboid, DOR ought to be equipped with its unique `dorID`.

One can relate to Qboid, as to a materialized meta-perspective (describing all of the integrated data with metadata), and the Integration View as an aggregated virtual meta-perspective (as a Qboid-based metadata view of the integrated data).

Covering Replication with Contexts One has also to discriminate between the replication terminology semantics in terms of both contexts. *Integration View* consider replica to be a

[34] E.g. in the case when the integrated, virtual record consists of many elements (vertical fragmentation) its `repo_id` will hold `VIRTUAL` value that would tag the structure as a container for all record attributes or tuples.

[35] For example, in context of Qboid, one can imagine two DORs representing the same record (case of replication), or a DOR containing two sub-DORs that cover complete set of record attributes (case of record vertical fragmentation). Those DORs can become part of *Integration View* context as a whole but also might participate in the Integration View only partially with their sub-DORs.

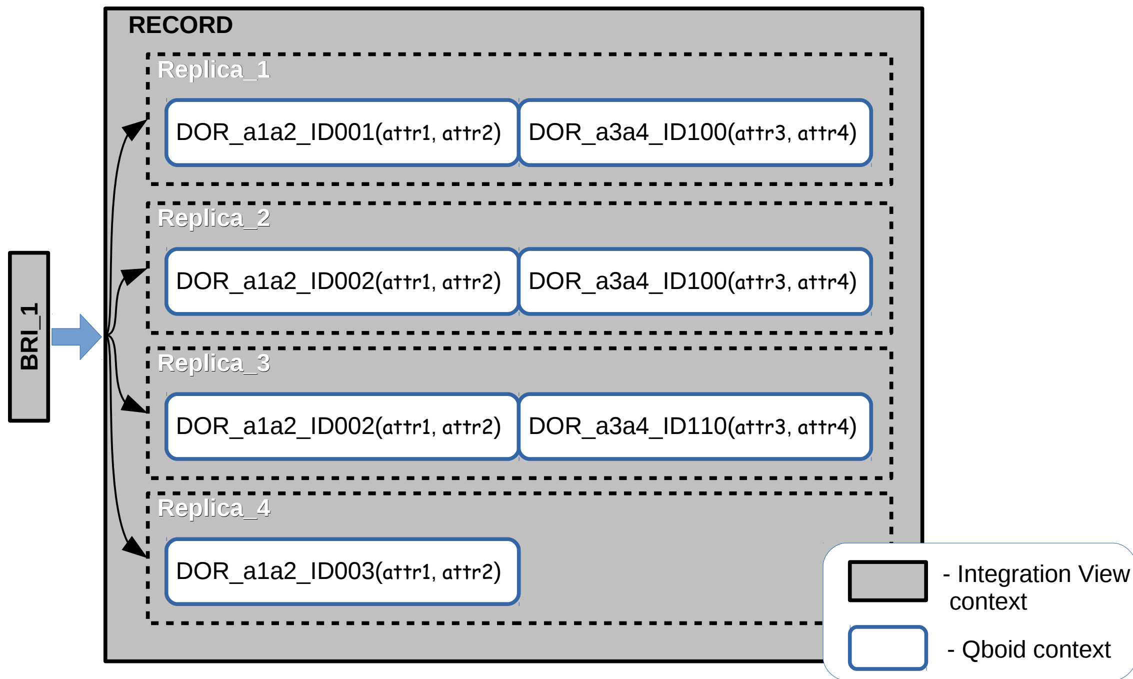


Figure 3.7: *Integration View* context uses Qboid DOR context, however both remain separate contexts. Distinct DORs with id: `001,002,003` describe the same set of attributes while sharing the same BRI

set of different (in terms of Qboid context – different DOR ids) DORs that describe the same *concept* [36] (e.g. two DORs describing `BRI_1`'s, `A` attribute), while in terms of Qboid, replica simply means replication of DOR that possibly is stored within different sources. This way the Qboid replication responds more to the native, physical data replication rather than *Integration View*-based replication by virtual schema definition [37] .

⋆ ⋆ ⋆

Let us now elaborate more on the `ContactDetails` and `ConnectionDetails` mentioned in Listings 3.8,3.9. Both pieces of information become an integral part of the rDOR structure, providing a single rDOR id key with multiple addressing details. They also provide the technical connection details for the dedicated optimization methods that require different connection schemes.

Listing 3.10: Contact and Connection Details of a rDOR

```
1  module ContactDetails{
2
3          typedef enum ProfileID{
4                  REGULAR, OPTI_INDEX, OPTI_OrderDependency, OPTI_MODEL[ ,...]
5          };
6
```

[36] For instance if one considers the *concept* of record, it has a common BRI. The records replication would thus mean that two or more DORs describe the same attribute(s) of this record. Each flowing DOR, describing (already described by another DOR) attribute, would be considered as a part of a new, separate replica.

[37] In other words, if we integrate two data sources, one of which had been created due to replication of other one – e.g. for recovery reasons (thus, it shares the data, scheme, etc.) – prior for the integration process, this replication will be covered by the Qboid context. If the data is represented in two independent stores with different store models, which can also involve different schemas, it is the *Integration View* that would be responsible for covering such replication pattern.

```
7         struct ConnectionProfile{
8                 ProfileID                         profile;
9                 sequence<binary>                  connectionDetailData;
10        };
11 };
12
13 module RegularConnectionDetail {
14        [...]
15
16        struct Details{
17                string                   host;
18                unsigned short           port;
19                CommunicationConf        protocolSpec;
20                ObjectBody               object;
21                sequence<AccessDetails>  objectView;
22        };
23 };
```

The `ConnectionProfile`, within the `ContactDetails`, contains the serialized binary data. This binary data is tagged by the `profile` field. The value of the `ProfileID` field will tell the system how the `connectionData` should be interpreted. Depending to what value the `profile` field is assigned, the connection details define the layout and the retrieval method of the referenced data. In a `regular` case the connection details are limited to host / port addressing tandem followed by networking configuration that can e.g. state the replicated DOR instances for load-balancing or fault tolerance purposes. Such a tagged profile approach enables the use of different contact policies defined by dedicated access optimization techniques.

Optimization Future Proofing This approach of specifying location and connection particularities provides some optimization future proofing. This is due to the fact that the first, `profile` field of the `ConnectionProfile`, relates to a way of accessing the data represented by the rDOR. When accessing the data in the "regular" i.e. no optimization involved way, the `ProfileID` would have to store the adequate value of `REGULAR` and point adequate (i.e. the *regular connection detail*) connection details sequence. In the case of a regular query, this would mean a simple data retrieval, based on what retrieval FAM method is being stated within FAM struct. On the other hand, while defining DOR that would be responsible for accessing the data with a dedicated optimization technique [38] , one would have to extend the `ProfileID` enumerated type with adequate value or use the already defined ones. Respectively, an additional new *connection details* definition – in the case of a new profile value – would have to be devised. The sequence of a binary represented `connectionDetailData` means that each DOR can have several sets of connection details. Each set, could later be used for accessing particular (described by DOR) data with one of the optimized ways, or the regular one.

In each data storing model, discussed in Chapter 2, the fastest way of accessing model basic chunk of data – regardless if it is a record, document, value, node etc. – was to use a kind of primary index – the *best record id* (BRI). BRI is also going to play a significant role in the design of Qboid and specifically the DORs. However, each BRI can not be recalled as a direct copy of the legacy data record BRI. It has to provide and maintain data records order in the represented virtual perspective that plays the role of the *integration view* context. It is also important because BRIs from different sources are not going to be forced to share common id scheme. This way the native BRI can be unique `long` value formed at one data source, while at the other it can be a set of attribute values that states the primary key, or a document ID – depending on the local model. To fulfil this requirement, BRI would have to be represented in a complex way, differentiating the *native BRI* from the *global BRI*, as depict in Listing 3.11.

[38] See Chapter 4 for detailed discussion on exemplary optimization techniques.

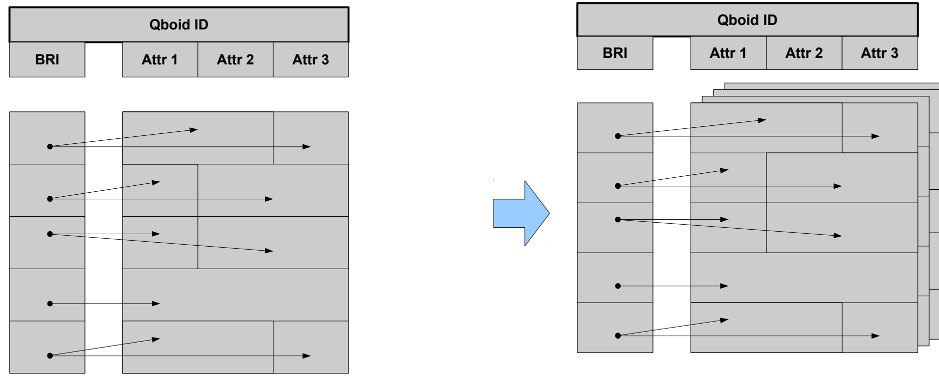
Listing 3.11: Virtual, BRI-based data identification strategy

```
CONCEPT{
        [...]
        typedef unsigned long GlobalBRI;
        [...]
};

REGULARCONNECTIONDETAIL {
        [...]
        typedef enum NATIVEFASTACCESSMETHOD{
                PK, OID, _ID, CompositeKey[, ... ]
        };
        NATIVEBRI{
                NativeFastAccessMethod nFAM_f;// flag for fam ie PK for
                    RDBMS or OID for ODB
                sequence<string>                bri;
        };

        FAM {
                sequence<NativeBRI>     nBRI;
                string  accessMethod;// query for RDBMS
        }:
        OBJECTBODY {
                [...]
                FAM             accessMethod;
                [...]
        };

        DETAILS{
                [...]
                ObjectBody              object;
                [...]
        };
};
```

To assure *regular* and unified data access, in general each connection detail should conform to the schema described with the example of `RegularConnectionDetail`. For complete listing of the metadata representing the *Qboid* and *Integration View* contexts representation see the Listing A.4.

Layered Planar Representation of Fragmentation Patterns A straightforward way of representing even the most complex fragmentation pattern is with a two dimensional matrix of building blocks. Those building blocks are record groups and single records – in terms of horizontal fragmentation – and the tuples and attributes in terms of vertical fragmentation. To build the cuboidal shape first one needs to provide a 2D matrix that can become the face of the Qboid. This matix would have to represent the complete – in terms of record definition of attributes – entity schema (see Figure 3.8(a)). This entity will be further represented by Qboid.

While considering replications one can imagine that the fragmentation pattern of this matrix is just one of many possible. Therefore if there would be additional replication of any of the entities schema part (like record, tuple attribute, group of records etc.) the additional matrix would have to be provided for representing this fact. Of course, each of the matrices can have a different fragmentation pattern and thus might be built on a puzzle basis according to the integration view designer intention and needs. That way a pile of matrices begins to form a cuboidal shape – namely the Qboid.



(a) Single complete Qboid entity definition mixed fragmentation pattern.

(b) Multiple complete Qboid entity definition mixed fragmentation pattern.

Figure 3.8: Complete Qboid entity definition covering single and multiple mixed fragmentation patterns.

Along this thesis each separate matrix representing the particular formula for completing the entity definition will be referred to as the *layer*. At the integration level, the complete layer representing the entity schema, is enough to answer any client call towards this entity, regardless of what particular records are requested, as long as they are part of the entity.

Slice the Dice – Representation of Replication The vertical representation of planar *layers* is responsible for covering the fragmentation patterns. However, Qboid is based on multiple layers representing the same entity. Since all layers share common schema (i.e. the entity definition) the Qboid can have numerous layers. However, each layer consists not only of physical fragmentation pattern elements, but conceptually it represents the concepts of the schema. Hence one has to be able to reach not only every replication fragment but also a complete schema element in the form of record, tuple or even a single attribute. For this purpose, as already mentioned, a global BRI has been introduced. The global BRI plays the role of a primary key that is however, independent of the fragmentation pattern, nor a local data particularities, including data's local BRI. This way one can combine corresponding records across the multiple *layers* without any prior preprocessing of the data local BRI. This is especially important due to the fact that the legacy data stores – whose data representation is partially being combined into the Qboid representation – are not forced to comply to any global schema. Therefore in this dissertation all records that share the common global BRI are referred to as a *slice*.
To visualize the intuition of the Qboid concept of a *slice* one should refer to Figure 3.9. This detailed description of each plain and slice is possible due to the BRI-based *key-value* definitions. The proposed model enables covering all kinds of fragmentations and replications in an arbitrary pattern, while preserving their unambiguous identification and access methods. The access methods are based on fast native querying methods supplied during each data source registration procedure from its *mediator* layer. This way every target data materialization procedure would not require live mediation and the query could be committed with the use of the low level database drivers.

The *Integration View* Context Building Blocks As mentioned in the previous section the construction of data structures in Qboid is based on the definitions. Let us consider the scheme for defining the data structures. The entire solution is going to be based on the key-value pairs to identify each part of data. The key would be the unambiguous (in adequate domain) ID of the object and the value would be varied depending on the actual type of the Qboid element. The

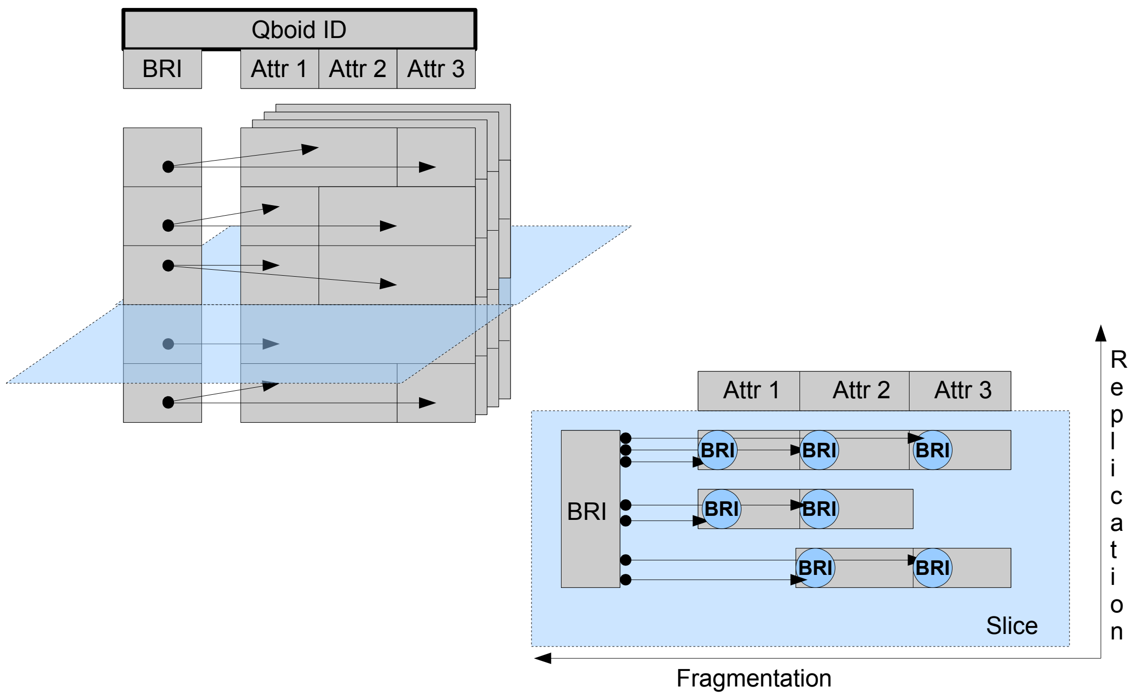


Figure 3.9: Slice represents all records that share common global BRI

description of this model will proceed from the most detailed pieces of the integration schema to the most general.

For the purpose of such solution there is a need to introduce the most basic structure representing the simplest piece of information in a global scheme – the *attribute* or a *cell*. Assuming the key-value pair approach the cell definition would have to consist of its unambiguous ID (in the domain of all the cells stored in the same record) and the data storing value (i.e. the way of accessing the cell or to be precise its reference) – rDOR.

Listing 3.12: Exemplary Cell Definition

```
1 cell{
2         key:        "name";
3         value:      rDOR_John
4 }
```

Next, in the structural order, there would be the *tuple*. It consists of a key name and the value, that is a map, containing an unlimited number of cells. In other word we could call it *SuperCell*.

Listing 3.13: Exemplary Tuple Definition

```
1 tuple{
2         key:                "emailAddress";
3         value:{
4                 user:       {key: "user";   value: rDOR_john30},
5                 domain:     {key: "domain"; value: rDOR_example_com}
6         }
7 }
```

To understand the reason for distinguishing between the cell and *tuple* we need to remember that these resources can be fragmented vertically. Therefore accessing a *tuple* may represent not only a logical entity but also a fragmentation pattern.

Following the structural design next level of complication stage would be a *record*.

Listing 3.14: Exemplary Record Definition

```
record{
        key:              "12";    //global BRI
        value:{
                name:{key:      "name"; value: rDOR_john)}, // name cell
                emailAddress:{                   //tuple named "emailAddress"
                        key:    "emailAddress";
                        value:{
                        user:   {key: "user";    value: rDOR_john30},
                        domain: {key: "domain"; value: rDOR_example_com}
                        }
                },
                age:            { key:  "age";   value: rDOR_30}// age cell
        }
}
```

Record will include explicitly a very important piece of information i.e. global BRI as its key value. The local BRI information is present in the implicit form in the DOR structure of cell or tuple. Record could be composed of cells or tuples however, all sharing the common BRI. By sharing the BRI, each cell/tuple would be classified as a component of the record. In this way all the cells/tuples in each record can be addressed owing to the same BRI. The value of the record is the sequence of cells /tuples. For less complex notation the cell/tuple names can be removed to form a pure key/value pair and replace the rDOR notation with the simple dummy reference providing the appropriate DOR object value for adequate BRI value (see Listing 3.15 for another example).

Listing 3.15: Exemplary Record Definition

```
13:{                                    //BRI value
    name: rDOR_DB1_13_name,
    emailAddress: {user: rDOR_DB2_13_user, domain: rDOR_DB3_13_domain},
    age: rDOR_DB1_13_age
}
```

Let us consider the exemplary DOR. The name DOR can assure the FAM for accessing all names in e.g. `DB1` using the native query. At this stage to address the appropriate record we must add `WHERE` operator defining the right BRI (i.e. the primary key in this case). In consequence, the DOR called here `rDOR_DB1_13_name` would include a query in the following (see also Listing 3.7) form:

Listing 3.16: SQL based FAM selection

```sql
SELECT name
  FROM   Emp
  WHERE  PK=13;
```

The three already mentioned elements of the data model however, will be referred only as the elements of a basic global scheme entity i.e. the layer. Layer is a view of a DB table content, composed of the distributed resources that registered their rDORs in central-point-of-reference. Layer is also key/value scheme dependent. As it is a container for records, it can be exemplified in the simplified notation as follows:

Listing 3.17: Qboid Layer

```
1  Users:{            // layer key
2          1..12:{                          // records keys
3             rDOR_DB1                                           // SELECT * FROM Emp;
4          },
5          13:{                                     // vertically fragmented
              record key
6              name: rDOR_DB1_13_name,        //SELECT * FROM Emp WHERE PK=13
7              emailAddress:{user: rDOR_DB2_13_user, domain:rDOR_DB3_13_domain},
8              age: rDOR_DB1_13_age
9          },
10         ...
11 }
```

By dint of the tuple idea, it became possible to overcome the vertical record fragmentation. However, in this way each remote cell / tuple / record from a DB server would have to be fetched by sending a single query per each. Therefore for instance requesting for four tuples from a remote server would cause need for sending and evaluating four simple queries one per each PK. Sending many simple queries could lead to major inefficiency along with increasing number of tuples per DOR. This problem could be solved by sending one query able to fetch all the requested records from one server at a time. However, this issue must be considered at the stage of Qboid design and should be a subject for carefull consideration by the designer.

Up to now the dedicated structures have simply considered the need for distributed heterogeneous integration. However, one last remaining issue has not been addressed yet in terms of data model i.e. the replication. To overcome the problem of replication we propose the idea of Slice. Qboid represents the definition of an entity, whereas the slice provides the definition of its records replications (see Figure 3.9). Each Qboid has to include at least one layer. Let us continue to exemplify the Qboid as the most general structure. Its key would be the name of a particular distributed entity resource Integration View. The value is represented as a map of layers. The layers, especially those representing the replications do not have to be complete in terms of their key definitions.

Listing 3.18: Qboid replica

```
1  Users:{                  //qboid key
2          Users:{          // layer key; layer 1
3                  1..12:{                      // record keys at node DB1
4                     rDOR_DB1                  // SELECT * FROM Emp;
5                  },
6                  13:{           // vertically fragmented row key
7                       name: rDOR_DB1_13_name,
8                       emailAddress:{user: rDOR_DB2_13_user,
                            domain:rDOR_DB3_13_domain},
9                       age: rDOR_DB1_13_age
10                 },
11                 ...
12         },
13         Users:{...           // layer key; Layer 2
14         }
15         ...
16 }
```

Moreover, to add the record replication information in the form of slice this structure needs to be enhanced with the additional DORs for the replicated data. This will be possible by simply placing a list of DORs representing particular data instead of single Database Object Reference.

Therefore assuming that the first 12 BRIs from the above example have four different, but data equivalent locations – those records can be represented as follows:

Listing 3.19: Qboid replication

```
1 Users:{                    //qobid key
2         Users:{            // layer key; Layer 1
3                 1..12:{                       // records keys at node DB1
4                    rDOR_DB1, rDOR_DB12, rDOR_DB7, rDOR_DB9
5                 },
6 ...
7 }
```

The selection of particular DOR can be based on numerous criteria like the first encountered, access or load balancing rules. This way each DOR based part of a snapshot can be accessed by using any of the DOR's replicas.
To express the database scheme in terms of Qboid the next level of integration can be introduced i.e. the *Keyspace*. The *Keyspace* would be then a container for Qboids representing the database scheme just as the Qboid is a container for all of the presented building blocks.

3.3.2.3 Interface Layer

The topmost layer in the integration architecture is the interfacing layer. It is responsible for direct client interactions. It involves client request handling by providing clients with the list of available integration views and accepting client integration-view-based calls for target data. Each response to client's request is preluded by data harvesting based on FAMs embedded within the Qboid based integration view metadata. Each metadata describing the data requested by the client contains a FAM that is committed toward the adequate legacy data source. The results are then being joined with use of global BRIs, also available owing to the metadata model.

In the case of unique queries this tends to be reasonable. However, in the case of queries that require large amounts of data to be collected and BRI-joined, or while the queries tend to repeat, the client interfacing adapter contains its local persistent cache that stores these particular query result sets. In that case if a client requests a data with a query that the result set has already been cached and persisted will be served with the cached result. Indeed such behaviour is inconsistency friendly due to the fact that the cache can store data that is outdated regarding the data from the legacy data sources. For this purpose there are two solutions.
The first is to simply commit a periodical update query that would refresh the cache content. This would, however, bring additional overhead to the system and cause unexpected inefficiency during the update periods, that could slow down other client calls. Additionally, it would have to lock every client request towards the cache content that is being refreshed. Therefore another solution has been provided.
At the stage of data modelling for the integration view each `ObjectBody` (a part of `ContactDetails`) contains a hash value. This hash value role is to check any changes that took place towards the data. The hash function can be custom, however, it always must consider the values of the data. In such case the data refreshing of the cached data, would first check if the calculated hashes of the data at the cache (Adapter) and at the local data source (Mediator) site are the same. Therefore the cache update procedure would only be commenced while the caches do not match. But even then, the update will deal only with the site whose hashes do not match those stored in the cached resource. This makes the refresh procedure work only on the hash mismatch and only for particular site data that has changed which makes it more effective and provides less system burden.

This way the refresh procedure is going to be committed in a lazy manner, and not as an eager periodical request.

Employing Patterns for Interface Layer The way the *Adapter* is going to handle a client request is a very important part of the architecture. This is mainly due to the fact that the integration architecture goal is to serve as a central-point-of-reference for big data sources, whose integration in the other way would be too expensive (millions and billions of dollars), or too complex, or even dangerous due to fragile or confidential data. While the *Mediator* and *Qboid* provide multi layered access policy, by configuring the access rights, it is also important for the client calls to work in a close to an in-time, or even live responsive manner. Achieving these two main aspects is considered. The first is how the data is going to be collected, which is the adapter back-end responsibility (based on FAM joining), the second – how the client calls are handled. Decoupling of the client request handling from adapter, is crucial due to the fact that each architecture appliance must be carefully designed and configured. Thus employing the strategy pattern for the front-end of the adapter allows to collect the data for client requests and respond with the use of independent architectural patterns. In case of tens of thousands (and even more) client requests the already mentioned patterns (see section 3.2.7) such as reactor, proactor, access object, etc. – would have to be used carefully depending on the integration architecture specific requirements and conditions considering the thread awareness, data retrieval duration etc.

3.3.3 Workflow

For complete understanding of the integration architecture, let us now follow its workflow.

As already discussed, the first step for each data source to be considered by central integration instance of Qboid, is to have a dedicated mediator instance. Additionally, each mediator prior to the function as a part of the architecture is thought to be registered. The registration process involves not only the network addressing information delivery of a data source, but also must equip the Qboid instance with *Contributory View* metadata (including FAMs). The local data source metadata generation rules are configured as a part of the mediator initialization procedure. The registration at Qboid site is focused around persisting the contributory view metadata for further utilization.

Having each source providing its contributory view, the Qboid designer can further design the global virtual integration schema – based on a set of registered contributory views – that will later be considered as a target for client calls. This integration views schema is independent of the legacy contributory views data and is virtually constructed only out of their parts. This way the integration view is also transparent regarding the legacy, contributory views.

A complete workflow is depicted in Figure 3.10.

Architecture client calls are assumed to be subjected towards an interface layer dedicated service in the form of an *Adapter*. The adapter service is responsible mainly for storing a full list of available integration views [39] . Each client call would require prior knowledge of what integration views are made available by Qboid. This way client may request particular integration view part by simply calling the adapter service API. Next the adapter is expected to untangle the FAM strings from the metadata information about the requested resource. Adapter then, sends the FAMs using the most lightweight and low level drivers towards each target data source. This way we assure that the querying process is maximally lightweight at each legacy site. The adapter awaits for each queried site to return its queries result set, and joins them using the global BRI acquired from the Qboid metadata prior to querying. This way the adapter stores the entire materialized integration view that is further sent to the requesting client as a response.

A more detailed UML sequence diagram of how the client call is processed is present in Figure 3.11 This figure considers also the usage of a heterogeneous index for distributed resources, that can be also defined using the integration view and can bring the optimization gains in pretty

[39] Also indexes. This will be elaborated more in Chapter 4.

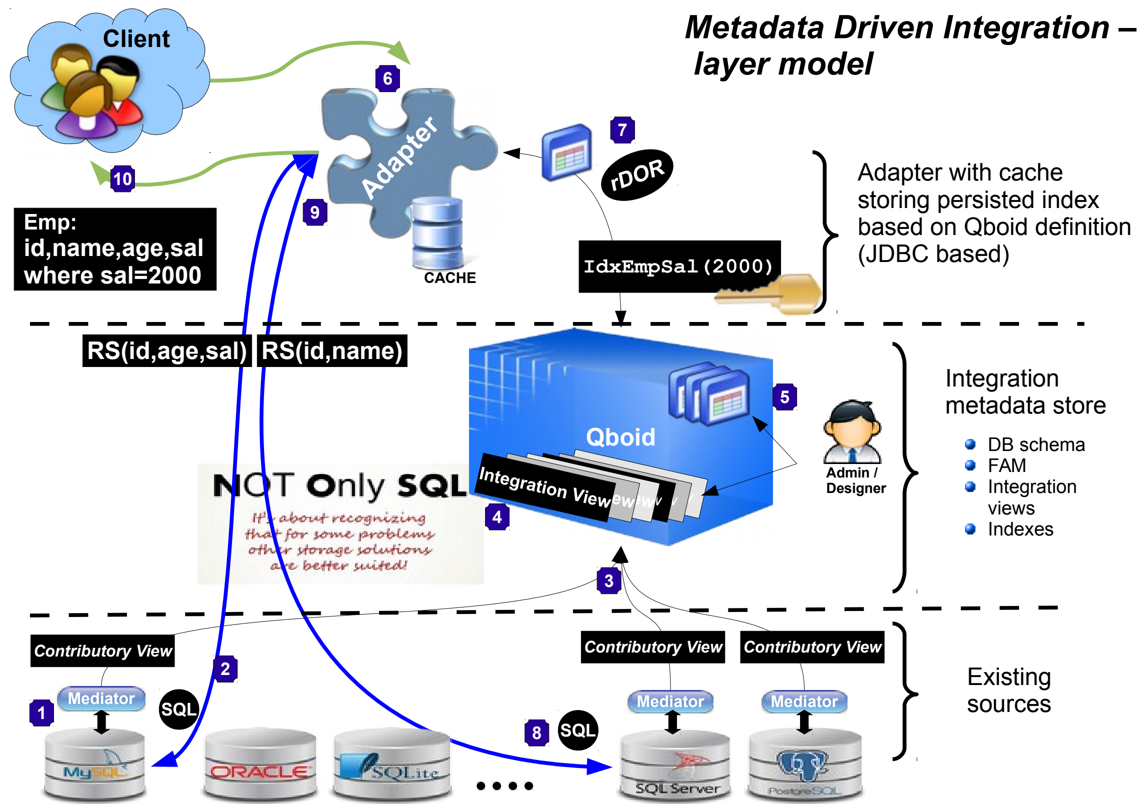


Figure 3.10: Metadata driven integration architecture workflow

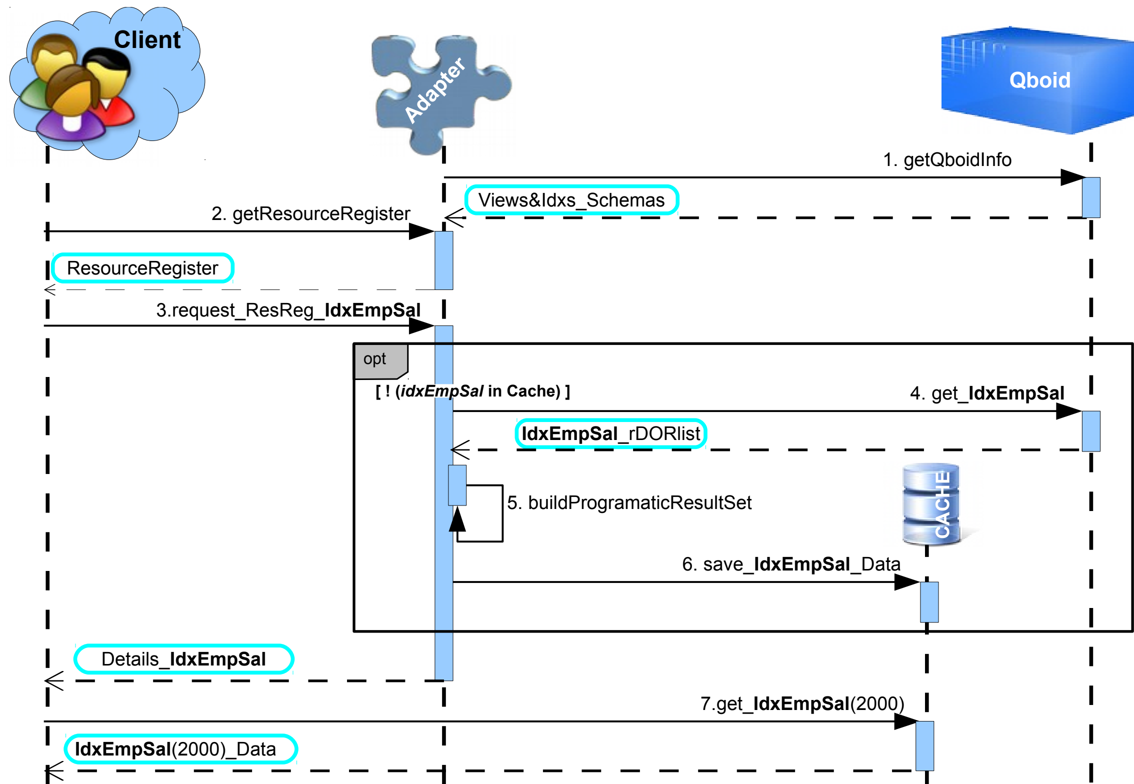


Figure 3.11: UML sequence diagram for client request processing life cycle

much the same way as the regular index does to the local database, but for distributed and heterogeneous data. This topic will be elaborated more in section 4.2.2 of the next chapter.

3.4 Faced Challenges

The process of integration distributed and heterogeneous data is a very challenging and complex problem. Throughout this chapter the presented solution has approached the most significant and at the same time challenging issues. The author presented an architecture that provides schema enabling the construction of the global schema that considers the data source wrapping. Owing to the layered mediator architecture, the architecture is also able to process the new source metadata by mapping between the sources and global schema based on the prior design and configuration. The communication schema based on the careful design of the monolithic schema enables providing means for expressing limitations in the mechanisms for accessing sources. Data extraction, cleaning, and reconciliation to the global schema can be done from the administrative level by carefully designed global, integration schema. Due to the simple mechanism – based on the hash function for each part of the global schema – it is possible also to configure the architecture to handle and process updates that can be propagated from the local to global schema. This way, a query expressed in terms of the global schema is reformulated in terms of (a set of) queries over the sources. The complete answer is prepared according to the partial sub-query results. The computed sub-queries are shipped to the sources, and the collected results are then assembled into the final response. The complex query plan guarantees completeness of the obtained responses.

The read-only nature of the proposed solution had to be assumed due to the enormous amount of complexity dealing with potential transaction handling that is out of the scope of this dissertation. However, the author believes that with a reasonable amount of time and resources, the architecture might also become write-only ready.

The global schema is modelled in a way that enables to map between the sources and the global integration view with the use of metadata. The architecture also provides a lightweight and low level means to answer queries expressed on the global schema with the use of native FAMs. Additionally the architecture future proofing goal was achieved by enabling a way for optimizing query answering with the profile approach in the introduced schema.

The topical architecture has also answered the problems of global schema modelling, data model, constraints, access limitations and provide data values representation regardless of schema or domain mismatches.

The provided functionality, however, is not automatic and must be a subject for very careful consideration prior to the function in an effective and desired way. As data modelling in the aspect of data integration is an art, it must also consider some future proofing and anticipate issues such as system load and networking and /or hardware limitations.

CHAPTER 4

Applications

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove."

— *Antoine de Saint-Exupery*

4.1 Integration

The topical integration architecture that is being discussed along the thesis is mainly aimed at providing an effective scaffold schema for building universal methods for accessing and / or gathering data from distributed and heterogeneous data sources. As it has already been discussed in the previous chapters it is not a trivial task. To overcome even the most basic integration issues, it is a must to concern and provide some complicated and extensive approaches and solutions.

As already mentioned in section 2.2.4 there are numerous patterns and mechanisms that have already emerged since the early beginning of the computer era. The mediation based, or federated solutions are both well known solutions since 80s of the twentieth century. Some other approaches have also emerged from the centralised to federated and distributed p2p approach. However, all those solutions had one common feature – they were all developed for past reality of low bandwidth internet connections and drastically smaller data. Modern, "big data-oriented" nature of the stored information however, requires more elastic and scalable solutions such as the architecture presented in this dissertation. Extensive possible applications for the topical solution require some in-depth discussion.

While all the features of the integration architecture have already been discussed in the previous chapter, a compare-and-contrast to the most recent research is essential to fully understand the current requirements and needs that other scientific teams have considered important to work on.

In the author's point of view, modern technologies must move forward from the 1990's situation – when enterprise used the data warehouses to combine polyseme stores – forward. During the last twenty years the DW and ETL tools were used for facilitating data extraction, transformation and loading. Customers were introduced to the data collected from multiple data sources with the use of data warehouses. However the important – integration – part had to involve multiple stages. Thus each execution includes:

- Extract – the parsing of the data source structure
- Transform – provide and apply transformation rules for common schema
- Clean – assure schema compliance
- Integrate – provide common schema
- De-duplicate – eliminate information redundancy

In modern enterprise appliances this process is very complex and expensive but also undoubtedly, a must. The data that were prepared in this way – while at the time of execution containing all requested, cleaned and extracted information – it loses many temporarily irrelevant information that soon might become important, but at the time of this execution dropped, due to not being requested. Therefore during each execution, much of the information is irreversibly lost, and thus requires an entire procedure to be repeated with the additional changes that consider new requirements in the case the dropped information occurs to be relevant in near future. For instance, collection data about employees might consider their ID numbers and full names while only the IDs are requested. Thus the full name information is then doped. One can easily imagine the situation that once the ID information is used, soon the full name information becomes the one that will be requested, but then the entire data collecting procedure must be repeated. This generates costs and it is time consuming.

The reason for data warehouses had been popular since the early 90s of the twentieth century is simple. Each time the data curation [1] from multiple sources is considered, almost always it requires data extraction, transformation and cleaning. This is mainly due to the fact that the data is completely independent and requires to be harvested for the requested information and transformed into the unified scheme. This procedure causes the non-requested data to be dropped. As an outcome, one obtains the clean information in the data warehouse. Later however, there is no point to return such information into the data source that it originally comes from, as it is already available in the data warehouse. What is more, the information after the data transformation becomes incompatible with the data source and thus would require reverse transformation. Finally, in most cases, cleaning and de-duplicating of the data makes it incomplete in terms of local data sources and thus, makes it impossible for reverse transformation.
This way the market for data warehouses is considered to be a billion dollar one, as it ties up every client to it.

However, the author believes that this situation will change in the nearest future. This is due to two main reasons. First of all, as already discussed in Chapter 2, the nature of the data changes from strictly *structured*, to the data that also contains the semi-structured data, time series, text data etc. In general, it is the data storing nature that must change and thus, the data requests would also become more, and more unstructured. The structured data tends to become the minority compared to the text, streams, JSON, etc. This is due to the fact that more aspects of life involve data storage. This, on the other hand, often forces to work with unstructured real time data. This is especially important in the areas of military, aviation or medicine where not only the data is important, but also the time that it arrives. A great example of modern application of a heterogeneous system might be the Mimic II. It was made for intensive care unit (ICU) that requires close monitoring, and as a result, a large volume of multi-parameter data is collected continuously [174]. There is an open access to historical data collected from the ICUs of Beth Israel Deaconess Medical Center from 2001 to 2008 and represents 26,870 adult hospital admissions (version 2.6). The system stores clinical data such as:

- typical structured data in the form of patients metadata like patient demographics, intravenous medication drip rates, and laboratory test results that were organized into a relational database (Postgres)
- semi-structured data – containing the prescription information
- text data – medical personnel notes (Accumulo [175])
- historical waveform data – the physiological waveforms, including 125 Hz signals recorded at bedside and corresponding to vital signs, stored in an open-source format (using SciDB for archived time series [176])

[1] In terms of management activities required to maintain data for long-term, in a way that information becomes available for reuse and preservation.

- real time data – non-historical, time series from the bedside monitoring devices (S-Store [177])

While this data is possible to fit the data warehouse solution, however as mentioned in Statement 1 there is no such thing as a universal model for all types of data, or as stated in [178] "one size fit all" dream. It is no longer applicable to the database market. Every time the the data nature must be confronted with the storage model that suits its storage best. Each model advantages are only effective towards a particular type of data, and thus must be used with extra caution considering the data nature.

This is where the integration architecture comes in. The number of data types and data storage engines is extensive. Thus solutions that would store numerous types of data would require the system administrator and users extensive knowledge of all disparate query languages. This situation is undesirable due to the high expert knowledge requirement, as well as numerous high error prone use cases. That is the spot where, and the reason why the architecture should replace the human user. For this purpose, one could consider one of the existing federated databases [2] . Even though the federated DBMSs work as a middleware over legacy DBMSs and provide effective interface for heterogeneous schemas, currently only the parallel DBMSs – which are limited to a single DBMS with partitioned, potentially replicated tables and a single schema – tend to be widespread in the enterprise.

This dissertation presents the third option. The proposed solution covers the best of the federations while eliminating their burden and overheads, and thus according to the author's research and opinion seems perfect for the job. As already elaborated in section 2.2.4 the problem with federations is that single failure of a component can cause an issue just in the same manner as a single point of failure in centralized architectures. What is more, the efficiency and latency of the federation is as good as its slowest component. The p2p nature of federation additionally brings the network connection burden and mapping processing. With dynamic mapping this becomes even more complicated.

However, the architecture presented in this dissertation, tends to take the best from the federated databases as the unified API, multiple kind of possible used data stores or lower client application complexity. Additionally, the Qboid central point-of-reference architecture provides possibilities for a single-, or multi-node distributed (e.g. Hadoop ecosystem implementation or Apache Accumulo [175]) implementation. What is more, it does not require complex distributed mapping sharing, as the network load is limited to schema transfer and updates while the transferred data is limited to pure metadata. Finally, the target data is reached only in a lazy manner which is more by native and low level usage of lightweight dedicated data sources drivers.

The most recent research (July 2015) that accomplishes similar tasks is the Intel Science and Technology Center for Big Data – BigDWAG project [21, 179].

4.1.1 *Polystores* as the Next-gen Federations vs Qboid-based Architecture for BigData Integration

From the performance point of view it is always a better choice to pick RDBMS for the structured data, Neo4J for highly correlated networking data, real time data into a stream processing engine, historical archives into an array engine or text into Lucene [180] engine or JSON into semi-structured NoSQL storages. This gives the final user the best performance-oriented grid. However, for effective usage of such an environment an abstract communication layer must be devised.

The introduced integration architecture goal is to provide such kind of functionality that can serve as a single-point-of-reference towards the legacy data sources it integrates. Additionally, it

[2] Such as the R*, Ingres*, Garlic, IBM's Information Integrator, etc.

provides the data model and location transparency. One can compare the topical architecture of this dissertation to the BigDWAG project. BigDWAG provides the abstraction – so called *island of information* – consisting of query language, data model, and that translates the global functionalities into the local data source dialect. The same way as the introduced architecture, the BigDWAG also assures the location transparency, and provides the same results for each query regardless of the data location. In contrast to Qboid based architecture however, each new local dialect requires separate *island*. Each data source can belong to many *islands*, just the same way as multiple Qboids can utilize the same data source within completely different integration schemas. On the other hand, the BigDWAG parses each query to AST, slices it into subqueries and involves each subquery translation towards the target data source query dialect (with the use of dedicated shim). Later the query execution is orchestrated along query engines and accumulates the results. Every "live" query manipulation in a real time system brings some overhead. In this case it brings some danger at the level of subquery generation and in the case of real time might cause latency due to badly optimized subquery, or too complicated syntax. The Qboid-based architecture advantage here is that each query is pre-computed and stored awaiting for execution towards the target data source. Additional prior testing might also be commenced in order to generate query profiling. Of course, this does not allow dynamic query manipulation, but in the case of careful query generation at the design stage it might bring a reliable and more efficient solution at cost of live query manipulation.

Both models allow comfortable usage of multiple query languages. Moreover, despite the fact of postulated *semantic completeness* [21] [3] in BigDWAG triggers, user-defined functions, and multiple notions of null are invariably considered system-specific and thus each *island* is forced to degenerate these functionalities. This is quite a disadvantage because such design does not permit to take full advantage of the local optimizers in the back ends query engines. Referring to Qboid, it cover neither transactions nor local capabilities at the Qboid central level, however this is due to the assumption that the stored queries already contain the detailed user-defined functions, adequate notions of `null` etc. that are specific to the local storage engine. This way one does not give up local DBMS functionality, while on the other hand leaving all back end processing to the local query engine.

The Qboid base architecture query processing is straightforward and does not provide additional "*shim*" processing, as in the case of BigDWAG polystores federation. Each query stored within Qboid is ready to be used in as-is basis with its target address details. While in the case of BigDWAG one has to provide additional casting expressing when an object should be accessed with a given set of semantics. This means simply that a relational query result to be joined with a NoSQL query would have to be a cast to the relational or NoSQL model. In Qboid the only factor is the BRI and Qboid definition that makes the results set in their correct place of the integration schema. The BigDWAG presents the syntactical approach that requires the client to express explicitly what kind of response model is expected. For example, in the query from Listing 4.1, client states that the query will execute in a relational scope in the federation, regardless of the actual data store model.

Listing 4.1: BigDWAG selection

```
1 RELATIONAL(SELECT *
2                  FROM R, CAST(A, relation)
3                  WHERE R.v = A.v
4 );
```

With Qboid such query would be executed as a result of a REST call procedure triggering a ready to be used, and persisted query that can be committed towards the adequate data source. That

[3] I.e. no loss of capabilities provided by underlying storage engines by adding them to architecture e.g. *polystore* or *qboid*.

way Qboid does not require additional syntax, nor prior user knowledge of the data storing engine particularities, and thus provides the heterogeneity transparency.

As a conclusion one should be aware, that the Qboid is not a federated class solution and therefore it is not forced for inter-node mappings nor *shimming* [4] . Moreover, it is a less complicated and mature project than the BigDWAG. Qboid is only a small, one-person project that obviously lacks the maturity and experience of Intel-supported BigDWAG project, involving such an esteem, respectable, and the well known leading database scientists such as Association for Computing Machinery's (ACM) A.M. Turing Award winner Prof. Michael Stonebraker. Both projects have been compared only on some basic functionality levels that aimed at showing the most modern and alike approaches for data integration. The Qboid class project cannot aspire or even be approached on even basis to BigDWAG due to their distinct nature, level of advanced problems considerations, or sophisticated solutions.

This way Qboid tends to support trend of the future data integration solutions that have become a topic for serious and extensive research from the largest enterprise parties (i.e. Intel) and leading database scientists and designers teams from mostly pioneering research institutions including MIT, Brown University etc.

4.2 Optimization

The traditional query optimizations are not able to support the distributed and heterogeneous environment of Qboid-based architecture. This is mainly due to their design and nature. There are several reasons why the classical approach will not fit this architecture. Let us mention, some of the most significant issues:

- The cost-based optimizers – each operation model must be maintained by the planner for the purpose of estimating the resource demand [27]. In the case of Qboid architecture it would mean that the global optimizer would have to understand the notion of each local operation across all integrated storage engines.
- Primitives semantics – Qboid does not provide semantics for understanding the nature of stored native query. The metadata content is required for describing the query purpose and nature. For instance matrix multiplications might be implemented as a grouping by aggregation – in the relational model whereas a distributed array store would use a set of scatter-gather operations for reaching the same goal. Accommodating such mismatch becomes a complex and non-trivial task.
- Considering a new data store – The optimizer would have to consider new circumstances, every time a new data store is "plugged" into the Qboid integrated architecture

The Qboid as an instance that holds only the query strings and their potential metadata might, however, store some additional information for optimization purposes. As already discussed in section 3.3.2.2 (namely the optimization future proofing paragraph), the Qboid metamodel is future proofed towards optimization purposes. Local storage engines may not provide such metadata, however, this is the part of the mediator to provide more sophisticated metadata-based knowledge from individual local sites. More details would depend on what optimization technique the local source data will participate in.

The local optimizer is not involved as far as the architecture is considered. It is assumed that the local optimizer can handle local query the way better than any other outer optimizer.

Apart from integration goal Qboid has also proven to be an effective and useful tool for applying various optimization techniques. Below the author discuss three examples of heterogeneous optimizations that are data model independent, or use the data model for optimization gains.

[4] *Islands* are like mediators that allow spanning over multiple models with shims – i.e. model-to-model translators.

4.2.1 Indexing Distributed and Heterogeneous Data

As already mentioned in sections 3.2.9, 3.3.2.1, 3.3.3 and also considered in Figures 3.4,3.10, the Qboid architecture in a straightforward manner might be considered as index oriented. As a matter of fact, the Qboid might be referenced to as a referential index for distributed, heterogeneous data location and retrieval.

In section 3.2.9 the key-value indexing nature was elaborated however, the optimization by indexing moves one step forward. This is mainly due to the fact that storing the BRIs, as *key* values, might become useful for reaching all the specified rDORs that conform to some projection (contain requested attribute values) or selection (acknowledge detailed selection conditions).

Inverted Index If an index book analogy is that for each *keyword* we get a list of referenced page number, then in the case of inverted index not only keywords are considered, but all *words* become keys.

Since indexing is the main workhorse for search engines they tend to enable the multiple [5] index pages scans. However, with this degree of magnitude it is not affordable for queries to compare each document in a one-by-one fashion because it would simply not scale for millions of queries being compared against billions of documents.

Indexes, is what makes search engines fast, and inverted indexes are used by virtually every search engine [6] . Due to high utilization of inverted index, search engines do not use RDBMS. This is caused by their specialized data structure and many specific optimizations.

A general principle behind the regular inverted index is mapping the key *values* to their location in files, documents or document collections. In the proposed architecture the idea is to store index that would map the values to a particular set of DORs that would be responsible for storing adequate records to the key value.

Since each concept of the integration schema can be accessed with a specified *key*, the *key* will also represent all instances that have been referred to it during the integration schema design stage. This is especially important when querying for data that is related to a particular *concept*.

4.2.2 Indexing Projections

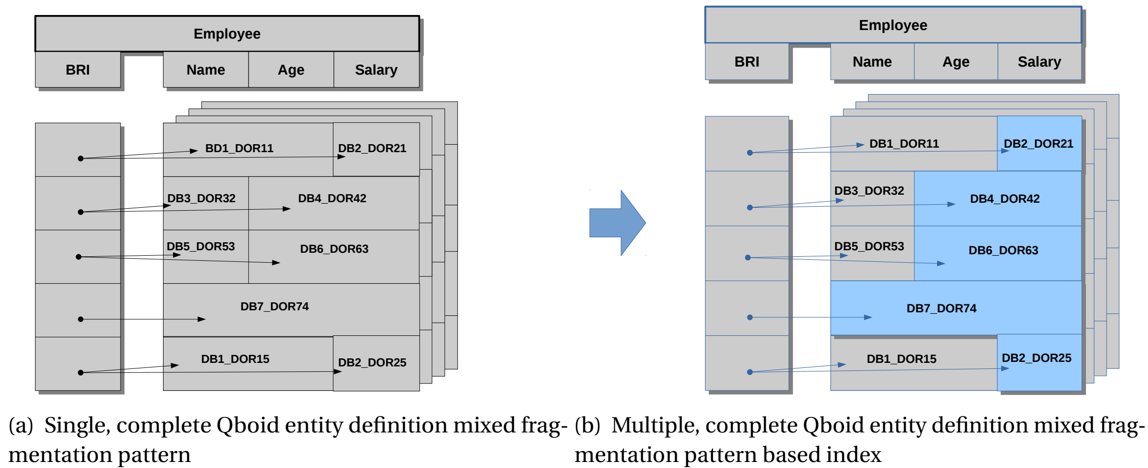
Let us assume a scenario that, based on the Qboid architecture, one needs to reach all values of the *salary* from integrated *Employee*. This would have to result in a set of values for *salary* originating from all registered storage engines that have been put into the global *Employee*'s schema design.

After receiving client request for creating a dense index on Employee according to its salary, a set of rDORs would be selected from the Employee's Qboid plain. Next they will be transformed into the index containing rDORs with the information about salary of each employee. Once the set of Emp instances storing salary information is known, each instance is investigated (owing to its particular FAM) for (BRI, salary) tuples. The collected information is now available as set of triplets (rDOR, BRI, salary). These results combined into one, three-column table will constitute an intermediary structure next transformed into the form of index.

In this particular case the index, where the `salary` is the *key* value, would have five rDOR reference values of `DB2_DOR21, DB4_DOR42, DB6_DOR63, DB7_DOR74`, and `DB2_DPR25`. Therefore the following fragmentation pattern will be created:

[5] Google scale of 20 billion of index pages

[6] Effectively finding all occurrences of a word in a document was the task that made Google start the MapReduce research.



(a) Single, complete Qboid entity definition mixed fragmentation pattern

(b) Multiple, complete Qboid entity definition mixed fragmentation pattern based index

Figure 4.1: Complete Qboid entity definition based index

Listing 4.2: Index on Employee's salary

```
<"EmpIntegrationView", // IntegrationView--ID
    <  "VIRTUAL", RECORD,
        <  0001       >
        <  >
        <  "DB1:HR:Empl", TUPLE,
            <  0001       >
            <  0x0011, True, MYSQL >
            <  // DOR DETAILS
            >
        >
        <  "DB2:HumanResources:Empl", ATTRIBUTE,
            <  0001       >
            <  0x0021, True, POSTGRES >
            <  // DOR DETAILS
            >
        >
    >
    <  "VIRTUAL", RECORD,
        <  0002       >
        <  >
        <  "DB3:Workers:Empl", TUPLE,
            <  0002       >
            <  0x0032, True, MYSQL >
            <  // DOR DETAILS
            >
        >
        <  "DB4:HumanResources:Empl", ATTRIBUTE,
            <  0002       >
            <  0x0042, True, POSTGRES >
            <  // DOR DETAILS
            >
        >
    >
    <  // third record with BRI = 3; alike record #0002
    ...
    >
    <  // fourth, complete record
    ...
    >
    <  "DB7:Pracownicy:Osoby", RECORD,
        <  0004       >
        <  0x0074, False, SELF,
        // DOR DETAILS
        >
    >
    <  // fifth record; alike the record #0001
    ...
```

```
48    >
49 >
```

What is left now is to group, distinct and sort those triplets by salary [7] . So the exemplary result reverse dense index would be as depicted in Listing 4.3

Listing 4.3: Index on Employee's salary

```
1  <idxEmpSalary, (
2     <1000, (          //the list of emp with salary = 1000
3        <BRI=0001, ref_to_DOR#0x0011, ref_to_DOR#0x0021 >,
4        <BRI=0005, ref_to_DOR#0x0015, ref_to_DOR#0x0025 >
5        ...
6     )>,
7     <1500, (
8        <BRI=0003, ref_to_DOR#0x0053, ref_to_DOR#0x0063 >,
9        <BRI=0004, ref_to_DOR#0x0074 >,
10       ...
11    >,
12    ...
13 )>
```

Having such inverted index will not only enable fast data access, but also direct access to the target storage engines.

Referring to Figure 3.10, and especially Figure 3.11 we have already placed there the cache instance. This is due to the fact that index, as an instance that potentially can be often used, might get persisted at the *adapter* interface site. This can be done in a lightweight and fast access NoSQL storage engine. Moreover, it can be placed as an in-memory store for faster access. This way the materialized index view of the requested data, can be effectively retrieved by the following index requests depending on the request and / or optimization policy-dependent configurations.

4.2.3 Exploiting Order Dependencies Optimization Technique for Qboid-based Integration Architecture

Qboid has also proven its usefulness for optimization techniques, that are not explicitly associated with its architectural design, as it took place with indexing. One of the most interesting techniques, that is being developed as a research project by IBM, Inc., on one of DB2's closed source experimental branches [181], is the *order dependency* based optimization.

4.2.3.1 Order Dependency Technique

The optimisation methods that exploit functional dependencies have already been known for decades.These methods are based on the fact of existence of injectivity property of the dependency function. What is important, is that if the domain of such a function is ordered, the function itself can preserve this order (i.e. is monotonic). This dependency was discovered and researched in [182–184].

It was noted that in the case of some specific columns – like dates – might define the monotonic function of an artificial primary key. In [182] the authors developed a simple method based on this observation. Its test proved an query execution increase in effectiveness of 20-50%. The remaining [181, 183, 184] papers abstract, so-called *order dependencies*, and present their proof theory similar to Armstrong's axioms. The promising result and enterprise attempts of IBM confirms the real value of this technique that soon might become part of master DB2 branch.

However IBM as a company focuses only on the DB2 codebase. Users of other DBMSs can not access them. Therefore the author has implemented similar optimisation mechanisms outside a specific database system with the use of Qboid integration architecture. We will use

[7] Here a MapReduce implementation is what could be used when considering large data volumes.

the Qboid architecture middleware mostly the same way the BigDWAG authors use theirs to manipulate the AST trees. However, the amount of computation targeted towards the query is minimal as it is limited only to query substitution (re-writing) based on configuration parameter (i.e. the `ProfileID` type) present in the global schema.

Order Dependency by Example Let us now show the motivating optimization potential and the value of order dependencies. We will discuss an exemplary schema depicted in Figure 4.2. If one considers a query for all sales within a time period, it could be stated as in the following

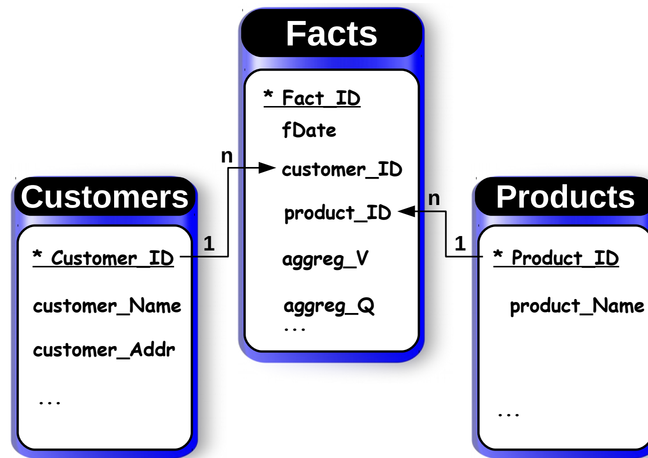


Figure 4.2: Exemplary database schema

Listing 4.4.

Listing 4.4: A query for sales in the indicated period

```
1 SELECT customer_Name , SUM( aggreg_V ) , SUM( aggreg_Q )
2         FROM Facts JOIN Customers USING ( customer_ID )
3         WHERE fDate BETWEEN '2008-12-13' AND '2008-12-15'
4         GROUP BY customer_ID , customer_Name
5 ;
```

One assumption is that the `fDate` column has no index. In such case the 4.4 query will require a full `Fact` table scan. Since the `Fact_ID` column of the `Fact` table is its primary key, thus all remaining columns functionally depend on it. In the case of the `f_Date` column the dependency function states as follows:

$$d : INT \rightarrow DATE$$

Implementation of an artificial primary key might be based on a *sequence* generator. In this particular scenario we can assume that the facts on sales from a particular day are recorded *after* all sales from the previous day. Thus, assuring that $d : INT \rightarrow DATE$ is non-descending.
For the particular query from Listing 4.4 we can assume that:

$$\begin{aligned} \texttt{xmin} &= min\{x; d(x) = \text{'2008-12-13'}\} \\ \texttt{xmax} &= max\{x; d(x) = \text{'2008-12-15'}\} \end{aligned}$$

and therefore, this query might become re-written to the equivalent query from Listing 4.5

Listing 4.5: A rewritten query for sales in the indicated period

```
1 SELECT customer_Name , SUM( aggreg_V ) , SUM( aggreg_Q )
2         FROM Facts JOIN Customers USING ( customer_ID )
3         WHERE Fact_ID BETWEEN xmin AND xmax
4         GROUP BY customer_ID , customer_Name
5 ;
```

The rewriting is based on the change in the WHERE clause.
This way the query from Listing 4.5 will utilize the range index on a primary key. This simple fact will make this query a much more time effective for execution than the pre-rewritten version. The assured monotonicity of the d function enables efficient computation of the xmin and xmax values, with the use of a well known binary search. The time overhead provided by the binary search, as proven in the following section, is notably smaller than the time saved by executing the optimized version of the example query. These observations have led to a rewrite algorithm that implements the idea presented above.

Order Dependencies Theory Assume a table T with its primary key P and remaining attributes $\{A_1,\dots,A_n\}$. Since P is the primary key of T there exists functions $f_1,\dots,f_n$ such that each tuple $(p,a_1,...,a_n)$ of the table T can be expressed as $(p,f_1(p),\dots,f_n(p))$. The existence of the functions $f_1,\dots,f_n$ validate the functional dependencies of the columns $A_1,\dots,A_n$ on the primary key P.

Assume that the domains of the columns P and A_i for a given $i \in \{1,2,\dots,n\}$ are linearly ordered sets. The functional dependency between P and A_i will be called an *order dependency*, if the function f_i is monotonic [8].

Such dependencies were initially called *monotonic dependencies* [182]. However, later inventors changed the name to *order dependencies*. The motivating example from the previous paragraph is based on such a dependency between the primary key Fact_ID and the column fDate.

4.2.3.2 Testing Order Dependency with Qboid Integration Architecture

The goal of the algorithm is to replace range conditions on the non-indexed columns to corresponding range search on usually indexed primary key. The algorithm is aware of the schema, and the order dependencies.

The tested solution can handle two types of queries: *select-project-join*, and the *grouping-aggregating* queries. It means that some general scaffold of the query would look as depicted in Listing 4.6

Listing 4.6: Query general schema

```
1 SELECT ...
2         FROM T JOIN T1 ON (T.f1k = T1.pk)
3          JOIN T2 ON (T.f2k = T2.pk)
4          ...
5         WHERE T.Ai BETWEEN  a1 AND a2
6         GROUP BY                  ...
7 ;
```

The WHERE clauses can also contain equalities and inequalities. In such case they are converted to atomic BETWEEN-based formula with the use of the same data type. The WHERE T.Ai =a selection is being converted to WHERE T.Ai BETWEEN a AND a.

As the first step, the algorithm identifies the fact table. Next it analyses the conditions in the JOIN ... ON clauses. The fact table connects other tables by foreign keys, while its primary key

[8] f_i is either increasing, non-increasing, non-decreasing, or decreasing.

is not connected by any other foreign key. In a query from Listing 4.6 the fact table is denoted by T.
In the cases including strings of foreign-primary key dependencies (e.g. the snowflake schema), the algorithm will also work. However, in the case of cycled dependencies, the algorithm will stop processing and return the original query without any further changes.

In the second step, the algorithm is required to check:

- if the `WHERE` clause references to a column of the identified fact table, and
- if this column has order dependency towards primary key

The third step involves `pmin`, `pmax` values search, in the case the f_i is non-decreasing:

$$\texttt{pmin} = min\{p; f_i(p) = \texttt{a1}\}, \quad \texttt{pmax} = max\{p; f_i(p) = \texttt{a2}\} \tag{4.1}$$

In the cases when f_i is non-increasing, analogously the algorithm will compute the `pmin` and `pmax` values as follows:

$$\texttt{pmin} = min\{p; f_i(p) = \texttt{a2}\}, \quad \texttt{pmax} = max\{p; f_i(p) = \texttt{a1}\} \tag{4.2}$$

Finally, the algorithm concludes replacing the `WHERE` with:

```
WHERE T.P BETWEEN pmin AND pmax
```

As already mentioned, since the function f_i is monotonic, the computation of `pmin` and `pmax` can be computed efficiently, e.g. using the binary search.

Implementation Characteristics The algorithm uses the Qboid as a middleware thus, the optimization takes place outside the target database system. The computation of `pmin` and `pmax` that satisfy conditions from (4.1) and (4.2) can be done in at least two ways. Both are based on the binary search.

In the first, a simple case is to send a series of queries in the course of the binary search. Its advantage is its inherent simplicity and the lack of any additional database object required. However, it causes numerous communication roundtrips with the database systems and thus, results in the additional system overhead.

Secondly, we can install appropriate stored procedures on the database side. This scenario is the one that has been accepted for implementing the binary search. When the optimizer on the Qboid middleware site is informed on the order dependency between the primary key and the column `fDate`, it will generate and install two stored functions. One of them shown in Listing 4.7 finds minimal `Fact_ID` for a given date.

Listing 4.7: PLSQL function that finds minimal Fact_ID for a given date

```
1  CREATE OR REPLACE FUNCTION get_min_find_by_date (
2          DF DATE
3   ) RETURNS integer AS $$
4  DECLARE
5      F INTEGER;
6      Z INTEGER;
7      S INTEGER;
8      D DATE;
9  BEGIN
10     SELECT MAX (Fact_ID) INTO Z FROM Facts;
11     S=1;
12     WHILE S<Z LOOP
13         S=S*2;
14       END LOOP;
```

```
15      F=S;
16      WHILE S>1 LOOP
17          S=S/2;
18          SELECT date into D from facts where Fact_ID = F - S;
19          IF D>= DF THEN
20              F=F - S;
21            END IF;
22
23        END LOOP;
24      RETURN F;
25  END;
26  $$ LANGUAGE plpgsql
```

An analogous function `get_max_fid_by_date(DATE)` that computes maximal `Fact_ID` is also needed. For the sake of readability the author have removed error handling code from the function `get..._fid_by_date(DATE)`. Potential errors might be caused by gaps in the numbering stored in the column `Fact_ID`.

Using the function from Listing 4.7 (and its twin `get_max...`), the optimizer will first issue queries for the corresponding margin values of `Fact_ID`. Then, it will put the collected parameters as values of bind variables in the modified query.

Unified Data Access Interface As already depicted in Figures 3.10 and 3.11, the client requesting calls towards the Qboid-based architecture is handled by the *adapter* instance from the interface, top layer. In the testing scenario the REST API has been used. The construct of the client REST API includes non optimized and optimized version of the query.

The client request will use the REST API to define whether the query response i.e. the result set, is going to be processed using a non optimized version of the query, or an optimized one. The author have prepared four implementations for data access layers. Two of them - using `JdbcTemplate` and `SimpleJdbcCall` – are *Spring Framework* [185] based, and the third is a pure JDBC connection. The final method gets the `pmin` and `pmax` hard coded. This is to compare the time of `pmin` and `pmax` retrieval and overhead that is brought by each of the three remaining methods.

The REST API is designed as follows. To retrieve unoptimized query answer the request url should like this:

```
http://localhost:8080/DIAS/rest/dbs/facts/2008-01-01:2008-01-02
```

Now, to request an optimized query depending on the query commuting mechanism the url would change to:

```
http://localhost:8080/DIAS/rest/dbs/facts/2008-01-01:2008-01-02/opti/X
```

Where *X* stands for the query commit method number. The *X* values were assigned as follows:

1. Spring `simpleJdbcCall` (stored functions)
2. Spring `JDBCTemplate` call statement (stored functions)
3. pure JDBC connection (stored functions)
4. Spring `JdbcTemplate` with (sub-queries rewrite)
5. hard coded `pmin`, `pmax` values

This way one can test each of the target optimization methods, in a simple and straightforward manner. The results are presented in the folllwowing paragraph.

Order Dependency Tests Results with Qboid-based Architecture It is required first to introduce the testing environment. The tests have been performed using the following hardware:

Table 4.1: Hardware configuration used for tests.

CPU	Intel Core i7-3612QM CPU @ 2.10 GHz x 8
RAM	15,6 GiB
Disk	SAMSUNG SSD PM830 2.5" 7mm 512GB
OS	Ubunty 14.04 LTS
Kernel	3.13.0-30-generic
Arch.	x86_64 GNU/Linux

The procedure was to measure response times for the REST client calls for optimized and unoptimized queries. This means that the test client called REST API that has used optimized or non-optimized query underneath, for the result set retrieval.

The tests have been performed using the software stated in Table 4.2:

Table 4.2: Software used in the testing process.

Java	java version 1.7_60 Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
REST Testing Client	ApacheBench, Version 2.3
Http Server	Apache Tomcat/6.0.29

Additionally, test cases assumed two optimization methods. One was to rewrite query with substitution of the `WHERE` clause, with two stored function results. For comparison reasons, the second case (fifth method) assumed replacing the stored functions with simple sub-queries to achieve the same goal as in the first case (see Listing 4.8). Namely:

Listing 4.8: Simple rewrite with sub-queries

```
1 SELECT Fact_ID , sum(aggreg_V) ,
2        FROM Facts
3        WHERE fDate BETWEEN (select min(Fact_ID) from Facts
4                      where fDate >= x)
5              AND (select max(Fact_ID) from Facts
6                      where fDate <= y)
7        GROUP BY customer_ID
8 ;
```

All tested use cases were conducted against the same request parameters and source data. The queried data range was between `2008-01-01` and `2008-01-02`. The result size was 31,546 MB. Each method was tested 50 times.

Measuring database response times for `pmin` and `pmax` was based on the Java's `currentTimeMilis()` method from `java.lang.System` [9] .

The test results are presented in Table 4.3.

The results have clearly shown that rewriting the `WHERE` clause boosts the target query almost four times. This is while only modifying the `WHERE` clause with sub-queries enabling the *primary key* in the role of index. This gives the idea of how order dependency based query can be effective.

[9]The detailed discussion for choosing this method has been conducted in [186]

Table 4.3: Test results for 50 request trials.

Activity	**Call Method**									
Document Length [MB]	31.546									
Method Name	**simpleJdbcCall**		**JdbcTemplate**		**JDBC**		**Subquery**		**Hard Coded**	**non-opti**
Stored Functions / Subqueries [ms]	*pmin*	*pmax*	*pmin*	*pmax*	*pmin*	*pmax*	*pmin*	*pmax*	0	
	13	14	7	6	5	6	7524	15493		
Avg.Time per request [ms]	44.010		29.762		27.196		23038.530		16.431	87681.945

Both of the queries do operate on `fDate` column that has not even been indexed. The result would be greatly better if only one would place an index on the `fDate` column. The hard coded column values for `pmin` and `pmax` are presented to compare the time performance of the query itself without the rewriting process.
Three remaining JDBC-based use cases for `(pmin,pmax)` retrieval are at worst three times slower than the hard coded `(pmin,pmax)` pair.
In general, the gained speed boost range from 87.681 seconds to only 0.027 seconds, which reduced the time of result retrieval for approx. 99,96%. Such gain for the discussed use case, is achieved with the best – pure JDBC – method, compared to the non optimized query.

⋆ ⋆ ⋆

As a result of order dependency testing scenario, the significant role of using Qboid as a middleware has been proven. Additionally, the usefulness of functional dependency – with a monotonic function with respect to linear ordering of domain – for query optimization, has also been confirmed. The proof-of-concept implementation of order dependency – that exploits order dependency on primary key – based on Qboid middleware made the solution vendor independent while focusing only on the query and the schema. The experimental results are promising and prove not only that the order dependency is effective while used for dedicated scenarios, but mainly justifies Qboid appliance as a middleware that can be used for vendor-independent query optimization techniques.

4.2.4 Polyglot Persistence as an Optimization Technique for Integration Architecture

Qboid, as a heterogeneous data integration and optimization placement middleware, additionally provides location transparent integration. This means that in an integration environment of heterogeneous data storage engines, Qboid can integrate data objects residing in some local sites. Moreover, Qboid provides the data objects notion to client application, in a way that does not require the application to cover the data object local particularities. This way the application logic remains clear and simple. Qboid also enables the use of data object replications that are present in the integrated storage engines grid. The main advantages of this functionality are: load-balancing, optimization, fault detection by verifying the data state and consistency [10] and

[10] At present Qboid is not responsible to assure the consistency across the replicas. This additional functionality can be provided with some extra global communication schema development.

storage model-driven data access. The system load shed can be assured by Qboids schema configuration (i.e. the `CommunicationConf`) to use adequate data replica according to the system load, request parameters, or accepted policies [11] . In the case of required complex analytical queries the system would be more responsive if more than just one object can be used to handle multiple queries. This is possible without the need to interfere with local DBMSs.

Optimizations can also utilize a data storage source metrics and estimators that explicitly are responsible for expressing the storage engine hardware and / or network capabilities. The fault or integrity check can easily be checked with hash value comparisons, as already mentioned in section 3.3.2.3.

However, in a heterogeneous data sources environment, Qboid provides a more interesting feature that will be focused on, in this section. This property, that is about to become a significant Qboid's virtue, is the ability to utilize the storage model on as-needed basis. This way it complies to the postulate of careful fitting of the data to the storage model – as stated in Statement 1.

Nowadays in a big project devotion to a single persistence mechanism usually leads to suboptimal architectures. However, architects of software systems are reluctant to use heterogeneous data sources. They are usually afraid of high cost of integrity maintenance. On the other hand, relational database systems are still perceived as universal storage solutions. However, the relational database model is devoted to process flat collections of business objects.

In recent years enormous proliferation (see Chapter 2) of numerous data storage engines, with their dedicated data model has provided additional possibilities for large systems.

Due to the abundance of task-oriented database systems architects face severe dilemma. The universality of relational databases allows modelling any application domain. However, a decision to use such a database as the only storage can negatively impact the performance. An interesting example of data causing such impact for relational databases is *graph data*.

4.2.4.1 Graph Model in Action

Querying the graph data stored in relational DBMS as been introduced to SQL with the use of SQL:1999 standard [187, 188] [12] in terms of *recursive queries*.

The goal of this third optimization technique with the use of Qboid integration architecture will be to justify its utility towards integrating data and optimizing its access with the use of heterogeneous data storage model. Qboid will allow exploiting the advantages of task-oriented database systems as they combine results returned by a number of databases into the form needed by an application. The proposed technique uses the Qboid integrator that abstracts the graph structure from a relational database and transfers it to a dedicated graph database (in this case Neo4j). Then relational queries to the graph data are mapped to the graph queries for appropriate identifiers of nodes. The result is then augmented with heavy relational data (remaining table attributes).

In following section the hybrid (graph-relational) method is prototyped with the use of Qboid framework [90, 189, 190].

The Use Case Let us assume a single table storing company employees according to the schema shown in Figure 4.3 This way each employee becomes a part of a tree hierarchy. One can easily imagine that each employee can have a twofold income. The first part based on the premium of his own work (e.g. sales) and the second part would be a fraction of the profits generated by his employees.

[11] E.g. Cycle of query redirection – i.e. the first data query redirected to `replica1`, the second query redirected to `replica2`, the third query redirected to `replica3`, and the fourth query again redirected to `replica1` etc.

[12] Implemented in [187, 188] by recursive *common table expressions* (CTEs) – i.e. a temporary named result set, derived from a simple query and defined within the execution scope of a `SELECT, INSERT, UPDATE`, or `DELETE` statement.

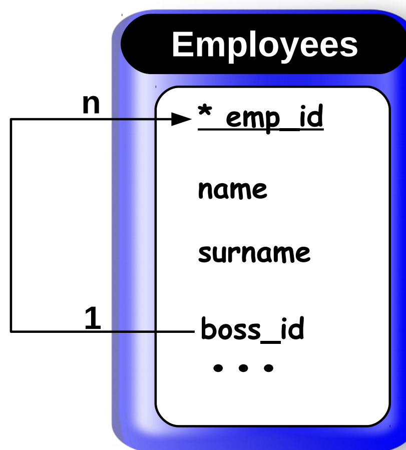


Figure 4.3: Exemplary table schema with the graph data

In terms of SQL one can generate the dependency tree graph with the use of `SELF JOIN` on `boss_id` and `emp_id` columns. In a use case scenario we assume that a query for *all employees in the tree spanned by a particular employee*, is often committed.
The first database management system to offer such query facilities was Oracle (since 1985). The Oracle's SQL dialect that would consider our use case scenario is exemplified in Listing 4.9.

Listing 4.9: A query for all employees under *Smith* in Oracle's early SQL dialect

```
1  SELECT * FROM Employees
2          WHERE level > 1
3          START WITH surname = "Smith"
4          CONNECT BY boss_id = PRIOR emp_id;
```

However, as already mentioned, the standard committee have decided to use CTE as a temporary named result set. The standard SQL:1999 syntax is therefore contained within Listing 4.10.

Listing 4.10: A query for all employees under *Smith*, according to SQL:1999 standard

```
1  WITH RECURSIVE emprec AS (
2                  SELECT emp_id, name, surname
3                          FROM Employees
4                          WHERE surname = "Smith"
5          UNION
6                  SELECT e.emp_id, e.name, e.surname
7                          FROM Employees e
8                                  JOIN emprec r
9                                  ON ( r.emp_id = e.boss_id )
10 )
11
12 SELECT *
13         FROM emprec
14 ;
```

The facilities to search graphs introduced into SQL:1999 significantly enriched the graph processing tools on the level of relational databases. An application programmer could formulate a single query where formerly a series of queries had been unavoidable. A performance review and comparison of existing implementations of the recursive queries in major relational database systems can be found in [191]. In modern application architectures that use object-relational mapping libraries the problem of querying graph data is even more complex [192–195].

Although relational databases implement recursive queries functionality and there is ongoing research on their optimization [196–198], however, relational databases only *support* graph queries. They do not implement them *natively*. A real efficiency improvement of big graph search is possible with the use of a dedicated graph storage engine. Neo4J [199] is one of such engines.

The efficiency of graph tasks in the dedicated graph databases is noteworthy higher than in the relational databases. Thus an architect is tempted to consider an implementation in which the graph structure is also stored in a graph database. Graph search is then partitioned into two subtasks. The first of them is the pure traversal among nodes. The second one augments the result of traversal with mass attributes retrieved from a relational storage. Note, however, that such a solution is notably more costly from a maintainer's point of view.

With the use of Qboid one can provide such an architecture in a totally transparent manner. Moreover, the Qboid middleware might be used to take care of the whole logic responsible for synchronizing a graph database, splitting queries and merging their results[13].

Testing Scenario In this section an effort will be made to prove that Qboid-based architecture can aid a significant improvement of big graph-based searches. As its additional virtue it needs to be noted that this process can be done transparently since a client application programmer is not aware of the actual data structure that is to be used. The querying in this use case scenario will focus on simple traversals among tree nodes.

Three similar testing scenarios have been devised. In all of them the Employees table schema from Figure 4.3 will be used. The target goal for each of the scenarios is to retrieve a fixed number of records as a result of a recursive query. Just as in the case of previous order dependencies test, the results are being consumed by client calls to the dedicated Qboid's adapter REST API.

To start testing a 18GB `Employees` table has been generated and loaded into PostgreSQL database.

The first testing scenario considered the use of `JDBCTemplate` [200] as a lightweight wrapper for the pure JDBC driver. Then using the REST API, multiple recursive queries have been commenced while resulting in approx. 8MB per 10 returned records.

The second scenario involved partial replication of recursive `Employee` table schema and data into separate table. Along the test this new table will be referred to as the `EmpBase`. This new table will contain the minimal useful set of attributes including the primary and foreign keys. This set of attributes is enough to construct the tree structure of the self referencing recursive hierarchy for the entire Employee table. Remaining attributes have no influence on this hierarchy. This way the same tree hierarchy of the `Employees` table schema can be build only according to the data stored in the `EmpBase` table schema. Calling the recursive query towards the *"light"*, replicated version of `Employees` table is assumed to build the hierarchy faster and thus simple joining with the rest of the attributes from the *Employee* table should give the same result in a shorter time.

The third test case considers a slight modification of the second scenario. This time however, the extracted minimal recursive schema information is moved outside the Postgres. The `EmpBase` in this case is being imported into a separate Neo4j graph database. Likewise the second scenario, the non-recursive and heavyweight attributes of the `Employee` schema stayed in PostgreSQL in the table `Employee`. To sum up, the entire schema and data is stored in PostgreSQL, and only the specific parts of the schema that describe the recursive relation are moved to the Neo4j. Since Neo4j implements recursive queries natively it is expected to provide additional performance boost. The migration process from PostgreSQL to Neo4J has been commenced with regard to official – Neo4J – recommendations [201, 202]. The parts of the Employees schema defining

[13] Due to the prototypical nature of the presented implementation not all those functionalities (like synchronization) are currently implemented. However, it will become a part of future research and extension of the Qboid based architectural idea.

recursive hierarchy have been presented as graph nodes' properties. Finally, the remaining heavy attributes left in the relational database are augmented to the data retrieved from Neo4j by means of the Qboid mapping. As a result, the entire recurrent hierarchy structure was built efficiently by Neo4j and then combined in Qboid with the attributes from PostgreSQL.
Qboid has provided significant functionalities. Due to the agile design, the client REST calls were transparent. Therefore, they enabled the client to focus on the delivered data and not the data delivering source. The client REST call tends to call for the Employee data regardless of the actual internal implementation of data structures.

Testing Results Prior to the test results one must be familiarized with the hardware testing architecture depicted in Table 4.4. Additionally, the used databases and framework information

Table 4.4: The hardware configuration used in the experimental evaluation

CPU	Intel Core i7-3612QM CPU @ 2.10 GHz x 8
RAM	15,6 GiB
Disk	SAMSUNG SSD PM830 2.5" 7mm 512GB
OS	Ubuntu14.04 LTS
Kernel	3.13.0-30-generic
Arch.	x86_64 GNU/Linux

are contained within Table 4.5.

Table 4.5: The software used in the experimental evaluation

Java	Java version 1.7_60 Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
REST Testing Client	ApacheBench, Version 2.3
Http Server	Apache Tomcat/6.0.29

The resulted response times were measured toward the REST calls in all three test scenarios. A testing REST client called REST API that used different ways of retrieving the data from the table `Employee`. Each REST call conformed to the following schema:

```
(...){strategy}/dbSchema/{dbEntityName}.json/limit={value1}&idoffset={value2}
```

The `strategy` variable has three possible values 1, 2, 3 depending on the test scenario. The `limit` and `offset` variables represented the number of resulted records and the offset just as SQL syntax.

The author commenced 1000 requests for each limited amount of result records. In the first test scenario the author have limited the number of requests to one because of long times needed to retrieve data. In two other scenarios we did not impose such a limit. The size of data for every response of 10 records was about 8 MB. The final results are depicted in Table 4.6

The test results have confirmed the expected performance boosts. The second test scenario has provided a noticeable performance boost compared to the clean PostgreSQL recursive query. Moreover, the third scenario has outperformed not only the first use case but also the second scenario occurred to be significantly slower.

⋆ ⋆ ⋆

This way Qboid can bring the best of each registered storage engine model. The testing involved development of an automatic method to integrate Neo4j with a relational database that stores

Table 4.6: The execution times of test queries related to the used data model

Request No	Result Records No	Total (Time per request- mean) [ms]
EmpFull (17GB-pgsql) (recursive)		
1	10	38 850,33
	100	188 414,76
	1 000	274 333,25
EmpBase (1GB-pgsql) + Widedata (17GB- pgsql)		
1000	10	1 783,06
	100	9 367,09
	1 000	17 069,04
EmpBase (1GB-neo4j)(recursive) + Widedata (17GB pgsql)		
1000	10	73,37
	100	774,33
	1 000	8 284,40

graph data. Secondly, testing concentrated on development of mapping, of recursive SQL:1999 queries onto a combination of a graph query and a simple relational query, followed by a proof-of-concept implementation of this mapping in Qboid with full transparency for an application programmer. On the whole, the established goal of proving the Qboid usefulness, has been achieved in the area of task-oriented data storage matching.

4.3 Conclusions

Throughout this the chapter author has conducted reasoning supported by actual testing scenarios that aimed at proving universal nature of Qboid based architecture. It has been tested that Qboid is a useful middleware framework for data integration [189] and optimization [90]. Additionally, one can consider using it while combining these both aspects.

The presented results are promising. Moreover, Qboid has provided full transparency for a client to get arbitrary data regardless of its original data source paradigm. Qboid has enabled fitting the data representation to the appropriate paradigm of data storage and its processing. The method to achieve this has been tested based on both SQL and NOSQL engines. This way Qboid tends to face SQL, NoSQL and NewSQL reality of modern data nature as a part of a wider *"Not Only SQL"* trend in the database development.

CHAPTER

5

Summary and Conclusions

The assumed goals and objectives from *Chapter 1 Introduction* have been achieved, accomplished and proven. The theses presented in this Ph.D. dissertation were proved to be true.

1. **Legacy data sources can be transparently integrated and interfaced with a Qboid based virtual meta-repository, while the data could still be gathered utilizing RESTfull service, without additional data manipulation, nor replication.**

The designed and implemented global integration schema for heterogeneous data sources allows generic and automatic metadata gathering. It enables a completely transparent integration for an arbitrary number of integration-participating data storage engines, based on central-point-of-reference metadata virtual (i.e. storing metadata instead of the real data) repository. The legacy, integrated data is referred, accessed and processed independently of its origin, due to metadata-oriented nature of the common communication and storage schema. This way each and every data operation (e.g. retrieval, access optimization)manipulates only the metadata instead of the target data. The metadata nature and the common schema, that the metadata is incorporated-in, provides the transparency and flexibility for various data access methods. The integrated data sources use the mediator-based architecture to wrap up every integrated data source and provide its metadata description in the form of contributory views. The actual top-level interface – that is being accessed by the integrated architecture clients – provides manually designed and configured global integration view that transparently represents the integrated data, based on the prior knowledge of contributory views'.

Each data source requires a dedicated – for its storage model – mediator, that will act as a middleware between the low-level storage engine and the central Qboid integrator instance. Mediator is responsible for supplying underlying data model schemas and native queries that later can be used to access the target data. The assumption behind this functionality, is that the local storage engine optimizer knows the best how to optimize its native query. This way the postulated solution does not tend to interfere with the local storage engine mechanisms, and takes the best out of its available optimization capabilities. Native queries are then embedded into the metadata sent to the Qboid that later persists it as a part of a global integration view. Those queries will then be sent by the integration architecture adapter instance directly towards the data source – using the most low-level access method available (e.g. JDBC driver) – on the client data request. Such design results in effective, lightweight and low-level data processing which – due to potentially large data volumes to be integrated from numerous, distributed data sources – was one of the major research goals.

2. **Well known optimization mechanisms and dynamic performance metrics can be implemented for Qboid based architecture, enabling optimized data access without interfering with the local data source potential optimization engine or its data schema.**

Since each data source can be accessed with its own native query, the query or data access algorithm can become optimized by: indexing, query manipulation techniques, or modifying the data access – accordingly to its model nature and available replicas. This is done owing to the flexible global integration schema that can be configured for an arbitrary optimization technique. Currently implemented and tested solutions involved three methods.
Indexing, as the first one, is obvious and provides optimization in a well accustomed way that does not require further explanation. The second, technique involves query rewriting. The author has discussed query rewriting based on the order dependency example, however it obviously can be extrapolated to virtually any other query optimization mechanism that would be based on query rewriting [1] across the integrated data source grid.
The last proposed optimization approach that leverage the Qboid-based architecture, involves configuring data access according to its nature. As the Qboid resolves the fragmentation and replication pattern puzzle effectively with its global integration view, one could use it for accessing data that can be stored in many places with different retrieval efficiency. Various retrieval efficiency metrics can originate from hardware / network performance, but also can leverage the data nature (e.g. recursive self relation) and use the most suitable storage model in the form of replica. In the case of heavy queries, or intense data source overload, the global view provides the load-balancing configuration option. However, prior to the load-balance usage, potential redundant replicas are expected to be integrated and contained within the global view.
Additionally, the access can be dependent also on estimations, metrics and cost model based on the configuration contained in global integration view schema. This provides the additional means to manipulate and balance system load depending on circumstances. As a conclusion, the outcome of the proposed solution has been proven to be reasonably effective with the use of the three test scenarios that can be generalized by analogy to the adequate optimization techniques accordingly.

5.1 The Limitation of Prototype and Further Works

The presented design of the Qboid-based integration architecture considers the data retrieval scenario – i.e. the read access. The motivation was mainly due to the fact that in the case of most big data environments, the data is supplied by the native local data applications and mechanisms that generate and fill the local data storing schemas. At the global level, the first desired step is to gather information for presentation. This is mainly for analytical and statistical reasons.
Write access, in some cases is simply impossible – as some of the legacy data sources are forbidden, or are not considered for the third party data interference and manipulations. Therefore, providing the additional functionality to write into local data sources from the top-level global integration view, would extend the scope of the work a couple of times and would not allow to focus on the dissertation theses. Hence, even though the write access functionality might be considered in terms of Qboid, in this dissertation it is considered out of scope.

The current implementation of the prototypical testing environment has been focused on proving the theses, and testing the optimization methods. The prototype provides heterogeneous data access for target data sources that played a role in testing scenarios. Since each data storage engine requires its own mediator implementation instance further mediator implementations can be introduced.

[1] In terms of global optimization one can consider SQL semantics duplicate elimination algorithms, use of the materialized views in the form of replicas instead of complex (e.g. joins, grouping, or aggregation of data) direct querying of the integrated data source, etc.

The mediator functionality for schema and / or native query generation is strictly data source dependent, however, it ought to be automatic based on dedicated low level solutions (e.g. ORMs). The contributory and global integration schemas storage can be arbitrary. This is due to the fact that the schema implementation is not relevant in terms of the architecture testing and theses. Therefore it has not been extensively considered. However, due to the complex structural nature of the schema, the data models that can handle deeply embedded structures – such as document stores (e.g. MongoDB), or graph stores (e.g. Neo4J) (in the cases relationships become a major aspect) – might be considered as an interesting topic for performance comparison research.
On the other hand, the global integration view, still has to be designed and implemented manually according to the global schema. The potentialy improved approach should consider automatic and / or *"intelligent"* global schema generation. Additionally, since current schemas consider only the entity-oriented integration, an evolutionary step might also consider extending it to support some kind of inter-entity relation handling. This seems to be fairly straight forward, possibly by referencing between the Qboid entity instances while using a kind of Qboid storage engine functionality (e.g. MongoDB referencing between objects, or Neo4j relationships). However, its efficiency and tuning require further research and testing.

The schemas designed and presented in this dissertation provided some future proofing. It involves the profile based approach for serializing / de-serializing of custom configuration options. The proposed appliances might also involve load-balancing, fault tolerance, integrity checking etc. It seems rather straightforward as the same mechanism has already been employed to handle configuration of the access optimization methods. There are also no barriers to extend those profiles with the additional, custom access configuration or optimization techniques, that can become a target for additional, future research.

Yet another topic, that can be substantial in terms of architectures efficiency, and that is not directly related to the basic architecture itself, is the process of combining the result sets from distributed sources to state the final result. The most basic way of the Qboid is to join the partial results from each site based on their best record ids. However, potentially interesting research can be commenced on queries that involve more complicated global queries than simple projections and selections.

5.2 Additional Mediator Functionalities

The implemented mediators have been focused on the data sources used in the testing environment. However, there is still a number of solutions, and legacy data sources that would require a dedicated mediator to be devised in order to participate in the Qboid-based integration architecture. One can consider mediators for the arbitrary NoSQL / NewSQL stores, or cloud storage – that would have to be specific towards each of the sources.

Due to read-only nature of the postulated architecture the mediator functionalities are passive towards the storage engine. However, still there is a considerable amount of issues to be researched – e.g. considering the dynamic mediator-to-datasource interaction involving generation of local optimization policies. This could, for example, include automatic new local index generation based on request, or system overload.

In current implementation, the potential load-balancing is based on hard coded, manual configuration. An interesting direction would be also to consider a live mediator-to-mediator communication that could help to provide live system balancing functionality. What is more, it could also consider some of the federation features in a way similar to the next-gen federation fashion – e.g in the case of BigDWAG .

Finally, the integration architecture seems an interesting concept to become implemented in a cloud friendly mean. Making the central integrating Qboid work in a cloud would bring some of the could benefits. However, one has to be warned that cloud is not a panacea for any domain

issue. Cloud is simply a tool that helps to outsource the infrastructure and make implementation cost effective, while still the target domain issues must be resolved by the developer.

APPENDIX

A Prototype Implementation

A.1 Integration Layer

To integrate multiple models it is required that the architecture will provide a form of unified semantic for describing contributory view. The proposed approach is to create values of C_SCHEMA and C_TABLE as stated in Figure A.1

Source Model	C_SCHEMA	C_TABLE
JDBC	• Database schema according to JDBC definition	• Any tabular schema structures such as table, view, etc.
OpenOffice	• Database schema according to JDBC definition	• Any tabular schema structures such as table, view, etc.
XML (SAX)	• A handle to the XML file. Always has default schema and virtual information schema.	• SAX: User defined XPath expression to concrete position in XML tree. Column, defining values of the rows in a C_TABLE, requires additional XPath for each C_TABLE.
XML (DOM)	• A handle to the XML file. Always has default schema and virtual information schema.	• DOM: virtual representation of recurring combinations of XML elements. C_TABLE generation requires manual or automatic "flattening" (denormalization) of "tables" e.g. using XSLT
CSV	• A handle to the CSV file. If the file does not exist, schema will be empty. Always has default schema and virtual information schema.	• File structure, based on the column name line.
MongoDB	• A MongoDB database.	• A MongoDB collection with a virtual table model, which represent the properties of the documents in the collection.
Cassandra	• Outer most grouping of the data i.e. Keyspace.	• Originate from ColumnFamily as a structure that contains a infinite number of rows grouped from Columns and SuperColumns
Salesforce	• A handle to the web services of Salesforce	• A type of Salesforce.com SObject.

Figure A.1: Contributory schema mapping

Here is an example of an XML (SAX) query retrieval procedure.

Listing A.1: Data hierarchy expressed with XML

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <root>
```

```
3        <employer type="governmental">
4                <cname>Company A</cname>
5                <employees>
6                        <employee>
7                                <ename>John Doe</ename>
8                                <gender>M</gender>
9                        </employee>
10                       <employee>
11                               <ename>Jane Poe</ename>
12                               <gender>F</gender>
13                       </employee>
14               </employees>
15       </employer>
16
17       <employer type="private">
18               <cname>Company B</cname>
19               <employees>
20                       <employee>
21                               <ename>Richard Roe</ename>
22                               <gender>M</gender>
23                       </employee>
24                       <employee>
25                               <ename>Mary Major</ename>
26                               <gender>F</gender>
27                       </employee>
28               </employees>
29       </employer>
30 </root>
```

If one desired to provide a contributory view as two separate C_TABLEs of `Employees` and `Employers` this would require some XPath expressions. It would involve defining the record scope and paths of individual values (in this context rather a column definition; see Listing A.2).

Listing A.2: XPath expressions definitions

```
1 <        employee ,  "/root/employer/employees/employee"                    //
     table employee(id , /ename , /gender)
2          <        ename ,   "/root/employer/employees/employee/ename"        >
3          <        gender ,  "/root/employer/employees/employee/gender"       >
4 >
5
6 <        employer ,        "/root/employer"
                                   //table employer(id , /cname , @type)
7          <        cname ,   "/root/employer/cname"   >
8          <        @type ,   "/root/employer/@type"   >
9 >
```

As the XML considered employer and employee to be related, an additional reference must be introduced. This can be done by easily providing extra field XPath expression for a foreign key to the employer id field. This way the employee definition might look as in Listing A.3

Listing A.3: XPath expressions definitions with foreign key

```
1 <        employee ,  "/root/employer/employees/employee"                    //
     table employee(id , /ename , /gender)
2          <        ename ,   "/root/employer/employees/employee/ename"        >
3          <        gender ,  "/root/employer/employees/employee/gender"       >
4          <        foreign ,  "/root/employer">
5 >
```

This way the target employee data would look as in Figure A.2.

ID	Cname	@type
0	Company A	governmental
1	Company B	private

ID	Ename	Gender	Foreign
0	John Doe	M	0
1	Jane Poe	F	0
2	Richard Roe	M	1
3	Mary Major	F	1

Figure A.2: Contributory View of two tables based on XML SAX XPath expressions

A.1.1 The IDL Scheme for Integration Contexts of Qboid and the Integration View

To understand the complexity of integration schema one should refer to Listing A.4 for a complete view. The context of Concept and Qboid has already been elaborated in section 3.3.2.2, however, not all schema content is clear and require more detailed explanation.

Let us focus on the content of the `RegularConnectionDetails` module. Its main part is the `Details` struct that covers the host / port network addressing but also contains three very important fields. First of them is the `CommunicationConf` responsible for defining the data retrieval behaviour in terms of its correspondence to the target data state at local data sources. In other words, it is responsible for defining rules for refreshing of the virtual data schema (e.g. BRI, or attribute change) according to local data changes. One can use the `verify` flag to force refresh if the data represented by the `Details` has changed (i.e. `ObjectBody`'s `objectValHash` changed) since its `last_update` timestamp. This verification is done with comparison of hash value that is calculated based on the hash function with arguments defined by `CommunicationSpec` accordingly to the data and schema described by particular `RegularConnectionDetail` with `ObjectBody` and `AccessDetails`.
The hash function formula is arbitrary, but should consider arguments that would define the data state. This way every time the function arguments (i.e. data state) change the hash value would be different forcing the architecture to repeat the procedure for collecting metadata information for such particular `RegularConnectionDetail`.

The second field type is `ObjectBody`. It is responsible for storing the hash value, mentioned in the previous paragraph, the FAM and the attributes list that are covered by the FAM. FAM, as already mentioned in section 3.3.2.1, stores enumerated type tag of `NativeFastAccessMethod` within the `NativeBRI` struct that defines by this the local data source BRIs and additionally, contains their actual sequence. Finally, FAM also contains the native `accessMethod` query string that serves each time target data retrieval is requested.

Finally, the `Details` struct contains a sequence of `AccessDetails`. This type stores use same tagging approach with the use of the `ConfigOption` self explanatory enumerate type to cover e.g. the replication types of *partial-* and *vertical- replica*. Additionally, the sequence of binary properties can be deserialized according to the value of the `ConfigOption` tag. The most basic role of `AccessObject` is to represent the partial replica (`P_REPLICA`) – which is considered in the cases the data described by a DOR is replicated at another site however, it does not cover entire global BRI related record, but only the same set of attributes as the current DOR or only its subset. On the other hand, the vertical replica (`V_REPLICA`) is responsible for storing a set of references to DORs that cover the same range of attributes as current DOR that are, however, vertically fragmented.
Apart from replicas `ConfigOption` can contain e.g. additional local data source login data or connection flags. For a detailed example please refer to the example in Listing A.4.

ConfigOptions can also be used to extend the details of the access policies. In such case, an additional enum type element would then have to be introduced with the additional struct type defined that can be later serialized into the AccesDetails sequence of binary properties. As each AccessObject is a part of a sequence each of the Details can contain a reasonable set of configuration options for a specyfic ContactDetails.

Listing A.4: Complete Scheme for Integration Contexts

```
module CONCEPT{

        typedef enum INTEGRATIONVIEWCONTEXT{
                ENTITY, RECORD, TUPLE, ATTRIBUTE[,...]
        };
        typedef unsigned long GlobalBRI;

        struct ACCESSOBJECT {
                string                          repo_ID;
                IntegrationViewContext          iv_Ctx;
                sequence<GlobalBRI>     gBRI;   // only for complete ENTITY
                    or RECORD
                sequence<Qboid::rDOR>    profiles;
                sequence<AccessObject> iv_Replicas;
        };

        struct INTEGREATIONVIEW {
                string                  iv_ID;
                sequence<AccessObject>  concept_AccessObject;
        };
};

module QBOID {

        typedef enum SOURCEID {
                SELF, POSTGRES, MS_SQL, MYSQL, ORACLE_11G, MONGO_DB
        };
        typedef unsigned long DOR_ID;

        struct RDOR {
                DOR_Id                  dorID;
                boolean                 vert_Fragm;
                SourceID                src_ID;
                sequence<ContactDetails::ConnectionProfile> object_Refs;
        };
};

module CONTACTDETAILS{
        [...]
        typedef enum PROFILEID{
                REGULAR, OPTI_INDEX, OPTI_MODEL, OPTI_OrderDependency, [...]
        };

        struct CONNECTIONPROFILE{
                ProfileID                       profile;
                sequence<binary>                connectionDetailData;
        };
};

module REGULARCONNECTIONDETAIL {
```

```
50      struct CommunicationConf {
51          boolean customSPEC;
52          // false - use some default settings for the profile
                protocol configuration options used to load balancing or
                integrity update check
53          boolean verify;         //true - check id data hash has
                changed since last interaction
54          date        last_update; //date of the last hash
                verification
55          [...]
56      };
57
58      typedef enum ConfigOption{      //vertical /partial replicas, data
            source, security configs etc.
59          v_REPLICA, p_REPLICA, SOURCE_CONF[, SECURE_CONF, "..."]
60      };
61          typedef sequence<rDOR> VReplicaConf;
62
63          struct P_Replica{
64              sequence<string> tuple_attr_names;
65              VReplicaConf tuple;
66          };
67          struct SourceConf {
68              string tag;
69              string value;
70          };
71      struct AccessDetails {
72          ConfigOption                config;
73          sequence<binary>            property;       ///depending
                on config deserialize the binary into e.g. VReplicaConf
74      };
75
76      typedef enum NativeFastAccessMethod{
77          PK, OID, _ID, CompositeKey[, ... ]
78      };
79      struct  NativeBRI{
80          NativeFastAccessMethod nFAM_f;// flag for fam ie PK for
                RDBMS or OID for ODB
81          sequence<string>            bri;
82      };
83
84      struct FAM {
85
86          sequence<NativeBRI>     nBRI;
87          string  accessMethod;// query for RDBMS
88      }:
89      struct ObjectBody {
90          sequence<binary>        objectValHash; //object value hash;
                CORBA uses 'unsigned long'
91          FAM             accessMethod;
92          sequence<string>            attributes;
93      };
94
95      struct Details{
96          string                  host;
97          unsigned short          port;
98          CommunicationConf       protocolSpec;
99          ObjectBody              object;
```

```
100                 sequence<AccessDetails> objectView;
101             };
102     };
```

A.1.2 The Integration Scheme in Action – Example

Let us focus on exemplary schema appliance The case scenario involves one employee Emp entity *Integration View* that is fragmented and replicated and thus requires a schema for integration. The fragmentation and replication patterns have been presented in Figure A.3 and A.4.

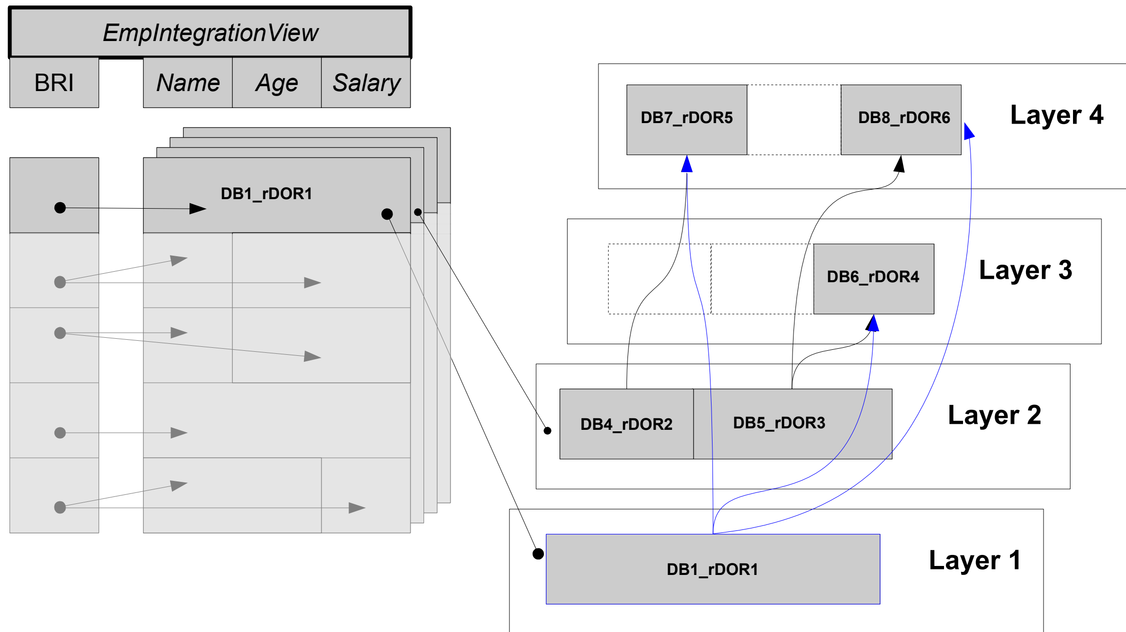


Figure A.3: *Integration View* of the Employee virtual schema. BRIs 1 to 100.

Listing A.5: IntegrationView Explicit Example for 1-101 BRIs

```
1  < "EmpIntegrationView", // IntegrationView--ID
2    < "DB1:HR:Employees", ENTITY, // AccessObject # 1 - a set of 1 to 100 BRI records
3      < 0001..0100 > // Global BRI #1-#100 of the entity
4      < 0x0001, False, SELF, // rDOR #1
5        < REGULAR, // ConnectionProfile #1
6          < x.x.x.1, 4444, // RegularConnectionDetail :: Details - The first : Layer 1
7            < False, False, '19.July .2014:16:30:00 ' > // protocolSpec
8            < !@##$%_HASH1_%$##@! , // ObjectBody
9              < PK,
10               < "1-100" >,
11               "SELECT * FROM Emp WHERE PK BETWEEN 1 AND 100;"
12             > // FAM
13             < "Name", "Age", "Salary" > // attributes
14           >
15           < V_REPLICA, // AccessDetails #1 - Layer 2
16             < 0x0002, False, POSTGRES, // rDOR #2
17               < REGULAR,
18                 < x.x.x.4, 4444, //DB4 see figure
19                   < False, False, '19.July .2014:16:30:00 ' >
20                   < !@##$%_HASH2_%$##@! ,
21                     < PK,
```

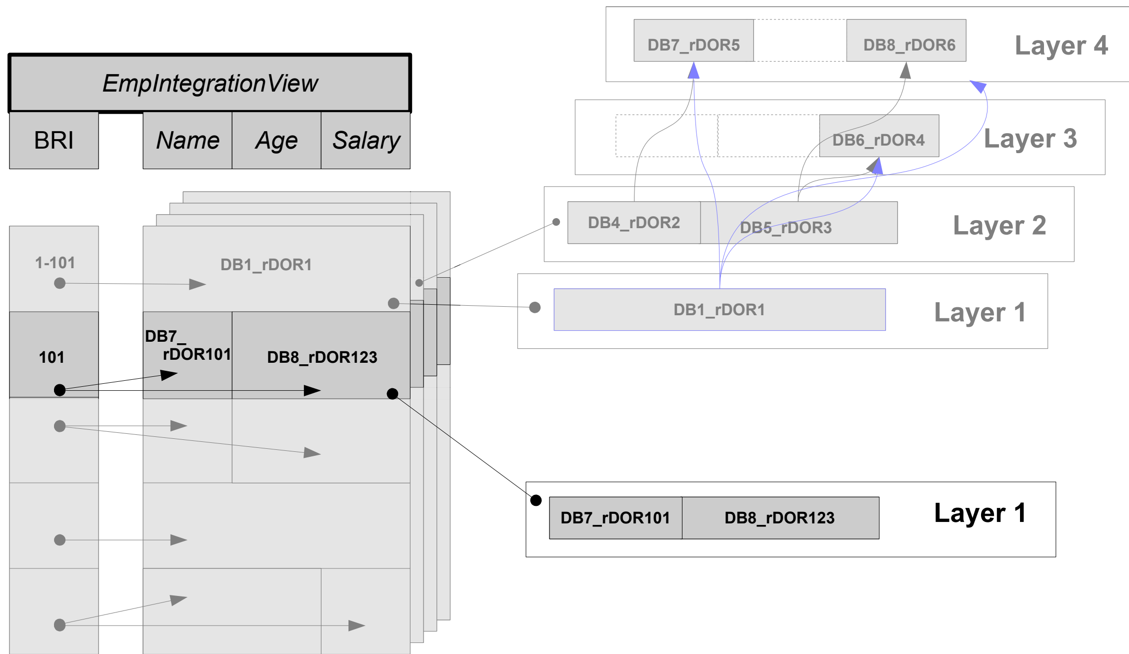


Figure A.4: *Integration View* of the Employee virtual schema. Record 101.

```
22                                  <  "1-100" >,
23                                  "SELECT name FROM Emp WHERE PK BETWEEN 1 AND 100;"
24                              >
25                              <  "Name"  >
26                          >
27                          <P_REPLICA,
28                              <  "Name" >
29                              <  0x0005, False, MONGO_DB,   // rDOR #5
30                                  <  REGULAR,
31                                      <  x.x.x.7, 4444, //DB7
32                                          <  False, False, '19.July .2014:16:30:00 ' >
33                                          <  !@##$%_HASH2_%$##@! ,
34                                          <  "_ID",
35                                          <"000000000000000000000001-000000000000000000000064">,
                                                // 12-byte hexadecimal ObjectId range
36                                          "db.emp.find ({
37                                              _id  :  { $lt :  ObjectId ("000000000000000000000064"),
38                                                        $gt :  ObjectId ("000000000000000000000001") },
39                                              {    _id :  0,  e_name:1      }
40                                          )"
41                                          >
42                                          <  "Name"  >
43                                          >
44                                      >
45                                  >
46                              >
47                          >
48
49                      >
50                  >
51              >
52              <  0x0003, False, MS_SQL,   // rDOR #3
53                  <  REGULAR,
54                      <  x.x.x.5, 4444, //DB5
55                          <  False, False, '19.July .2014:16:30:00 ' >
56                          <  !@##$%_HASH3_%$##@! ,
57                              <  PK,
```

```
58                                          < "1-100" >,
59                                          "SELECT age, salary FROM Emp WHERE PK BETWEEN 1 AND 100;"
60                                      >
61                                      < "Age", "Salary" >
62                                  >
63                                  <P_REPLICA,
64                                      < "Salary"        >
65                                      < 0x0004, False, CASSANDRA, //rDOR #4
66                                          < REGULAR,
67                                              < True, False, '19.July .2014:16:30:00 ' > // true - use for
                                                  load balancing
68                                              < // here define load balancing configuration
69                                              >
70                                              < !@##$%_HASH4_%$##@! ,
71                                                  < CompositeKey,
72                                                      < " PartitionKey =6", "ClusteringKey =1-100" >,
73                                                      "SELECT salary FROM Emp WHERE PartitionKey=6
74                                                          AND ClusteringKey >= 1 AND ClusteringKey <= 100;"
75                                                  >
76                                                  < "Salary"      >
77                                              >
78                                              < // no replicas in schema; the DOR#6 not considered for
                                                  load balancing purpose as DOR#4
79                                              >
80                                          >
81                                      >
82                                  >
83                                  <P_REPLICA,
84                                      <0x0006, False, ORACLE_11G,  //rDOR #6
85                                          <REGULAR,
86                                              <True, False, '19.July .2014:16:30:00 ' > // true - use for
                                                  integrity check
87                                              < // here define integrity check configuration
88                                              >
89                                              < !@##$%_HASH6_%$##@! ,
90                                                  < PK,
91                                                      < "1-100" >,
92                                                      "SELECT salary FROM Emp WHERE PK BETWEEN 1 AND
                                                          100;"
93                                                  >
94                                                  < "Salary"      >
95                                              >
96                                              < // no replicas in schema; no other replica considered for
                                                  integrity check purpose
97                                              >
98                                          >
99                                      >
100                                 >
101                             >
102                         >
103                     >
104                 >
105                 < P_REPLICA, // AccessDetails #2 - Layer 3
106                 // the DOR #4; blue arrow
107                 >
108                 < P_REPLICA, // AccessDetails #3 - Layer 4
109                 // the DOR #5 and $6; blue arrow
110                 >
111             >
112         >
113     >
114 >
115 < "VIRTUAL", RECORD, //AccessObject #2; VIRTUAL - means vertically fragmented ctx element, here 'record'
116     < 0101        >
117     < //Empty - as a VIRTUAL this record is a container that covers vertical fragmentation of 0101
          record out of attribute and a tuple    >
118
119     < "DB7:HR:Empl", ATTRIBUTE, // Access Object # 3 e.g. another set of records (entity), a single
          record, a tuple or an attribute ; starting form gBRI 101
120         < 0101        >
121         < 0x0123, True, MYSQL > //rDOR #123; True - here means that complemented by following
              AccessObject for complete record
122         < REGULAR,
```

```
123                <  x.x.x.7, 4444,
124                    <  False, False, '19.July .2014:16:30:00 '   >
125                    <  // no custom communication spec.     >
126                    <  !@##$%_HASH123_%$##@! ,
127                        <  PK,
128                            <  "101"   >,
129                             "SELECT name FROM Emp WHERE PK = 123;"
130                        >
131                        <  "name"   >
132                    >
133                    <  // no  replicas  in  schema;
134                    >
135                >
136            >
137
138        >
139        <  "DB8:HR:Empl", TUPLE, // Access Object # 4 – compliment the Access Object #3 to complete record
140            <  0101        >
141            <  0x0234, True, neo4j > //rDOR #123
142            <  REGULAR,
143                <  x.x.x.8, 4444,
144                    <  False, False, '19.July .2014:16:30:00 '   >
145                    <  // no custom communication spec.     >
146                    <  !@##$%_HASH243_%$##@! ,
147                        <  ID,
148                            <  "101"   >,
149                             "  MATCH (emp) where emp.id = {  234  }
150                                RETURN emp.age, emp.salary; "
151                        >
152                        <  "Age,  Salary "     >
153                    >
154                    <  // no  replicas  in  schema;
155                    >
156                >
157            >
158
159        >
160    >
161    <... // Next records
162    >
163 >
```

APPENDIX

B

Standards and Classifications

Name	Focus	Description
DDI	Archiving and Social Science	The Data Documentation Initiative is an international effort to establish a standard for technical documentation describing social science data. A membership-based Alliance is developing the DDI specification, which is written in XML.
EBUCore	The EBUCore metadata set for audiovisual content	EBUCore is a set of descriptive and technical metadata based on the Dublin Core and adapted to media. EBUCore is the flagship metadata specification of EBU, the largest professional association of broadcasters around the world. It is developed and maintained by EBU's Technical Department. EBU has a long history in the definition of metadata solutions for broadcasters. EBUCore is registered in SMPTE. It is also available in RDF and the documentation.
EBU CCDM	The EBU Class Conceptual Data Model - CCDM	'The EBU Class Conceptual Data Model (CCDM) is an ontology defining a basic set of Classes and properties as a common vocabulary to describe programmes in their different phases of creation from commissioning to delivery. CCDM is a common framework and users are invited to further enrich the model with Classes and properties fitting more specifically their needs.
FOAF	Friend of a Friend (FOAF)	The Friend of a Friend (FOAF) project is about creating a Web of machine-readable homepages describing people, the links between them and the things they create and do.
EAD	Archiving	Encoded Archival Description - a standard for encoding archival finding aids using XML in archival and manuscript repositories.
CDWA	Arts	Categories for the Description of Works of Art is a conceptual framework for describing and accessing information about works of art, architecture, and other material culture.
VRA Core	Arts	Visual Resources Association – the standard provides a categorical organization for the description of works of visual culture as well as the images that document them.
Darwin Core	Biology	The Darwin Core is a metadata specification for information about the geographic occurrence of species and the existence of specimens in collections.
ONIX	Book industry	Online Information Exchange - international standard for representing and communicating book industry product information in electronic form.
CWM	Data warehousing	The main purpose of the Common Warehouse Metamodel is to enable easy interchange of warehouse and business intelligence metadata in distributed heterogeneous environments.
EML	Ecology	Ecological Metadata Language is a specification developed for the ecology discipline.
IEEE LOM	Education	Learning Objects Metadata - specifies the syntax and semantics of Learning Object Metadata.
CSDGM	Geographic data	Content Standard for Digital Geospatial Metadata maintained by the Federal Geographic Data Committee (FGDC).
ISO 19115	Geographic data	The ISO 19115:2003 Geographic information -- Metadata standard defines how to describe geographical information and associated services, including contents, spatial-temporal purchases, data quality, access and rights to use. It is maintained by the ISO/TC 211 committee.
e-GMS	Government	The e-Government Metadata Standard (E-GMS) defines the metadata elements for information resources to ensure maximum consistency of metadata across public sector organizations in the UK.
GILS	Government/ Organizations	The Global Information Locator Service defines an open, low-cost, and scalable standard so that governments, companies, or other organizations can help searchers find information.
TEI	Humanities, social sciences and linguistics	Text Encoding Initiative - a standard for the representation of texts in digital form, chiefly in the humanities, social sciences and linguistics.

NISO MIX	Images	Z39.87 Data dictionary - technical metadata for digital still images (MIX) - NISO Metadata for Images in XML is an XML schema for a set of technical data elements required to manage digital image collections.
<indecs>	Intellectual property	Indecs Content Model - Interoperability of Data in E-Commerce Systems addresses the need to put different creation identifiers and metadata into a framework to support the management of intellectual property rights.
MARC	Librarianship	MARC - MAchine Readable Cataloging - standards for the representation and communication of bibliographic and related information in machine-readable form.
METS	Librarianship	Metadata Encoding and Transmission Standard - an XML schema for encoding descriptive, administrative, and structural metadata regarding objects within a digital library.
MODS	Librarianship	Metadata Object Description Schema - is a schema for a bibliographic element set that may be used for a variety of purposes, and particularly for library applications.
XOBIS	Librarianship	XML Organic Bibliographic Information Schema - a XML schema for modeling MARC data.
PBCore	Media	PBCore is a Metadata & Cataloging Resource for Public Broadcasters & Associated Communities
MPEG-7	Multimedia	The Multimedia Content Description Interface MPEG-7 is an ISO/IEC standard and specifies a set of descriptors to describe various types of multimedia information and is developed by the Moving Picture Experts Group.
MEI	Music notation	Music Encoding Initiative is a community-driven effort to create a commonly accepted, digital, symbolic representation of music notation documents.
Dublin Core	Networked resources	Dublin Core - interoperable online metadata standard focused on networked resources.
DOI	Networked resources	Digital Object Identifier - provides a system for the identification and hence management of information ("content") on digital networks, providing persistence and semantic interoperability.
ISO/IEC 11179[8]	Organizations	ISO/IEC 11179 Standard that describes the metadata and activities needed to manage data elements in a registry to create a common understanding of data across organizational elements and between organizations.
ISO/IEC 19506[9]	Software Systems	ISO/IEC 19506 Standard called Knowledge Discovery Metamodel is an ontology for describing software systems. The standard provides both a detailed ontology and common data format for representing granular software objects and their relationships enabling the extractions such as data flows, control flows, call maps, architecture, database schemas, business rules/terms and the derivation of business processes. Used primarily for legacy and existing systems security, compliance and modernization.
ISO 23081[10]	Records management	ISO 23081 - three-part technical specification defining metadata needed to manage records. Part 1 addresses principles, part 2 addresses conceptual and implementation issues, and part 3 outlines a self-assessment method.
MoReq2010	Records management	MoReq2010 - A specification describing the MOdel REQuirements for the management of electronic records.
DIF	Scientific data sets	Directory Interchange Format - a descriptive and standardized format for exchanging information about scientific data sets.
RDF	Web resources	General method for conceptual description or modeling of information that is implemented in web resources, using a variety of syntax formats.
MDDL	Financial market	The (Financial) Market Data Definition Language (MDDL) has been developed by the Financial Information Services Division (FISD) of the Software and Information Industry Association (SIIA). MDDL is an extensible Markup Language (XML) derived specification, which facilitates the interchange of information about financial instruments used throughout the world financial markets. MDDL helps in mapping all market data into a common language and structure to ease the interchange and processing of multiple complex data sets.
NIEM	Law enforcement; Social services;	NIEM—the National Information Exchange Model—is a community-driven, US government-wide, standards-based
	Enterprise resource planning	approach to exchanging information. NIEM's data domains are growing standards developed and maintained by domain communities. These are just some sample domains included or being developed in NIEM: Justice; National Security Intelligence; Biometrics (for Law enforcement); Emergency Management; Security Screening; Human Services; Children, Youth, and Family Services; Health Services; Immigration; National Infrastructure Protection; Government Resources Management.
SAML	Shibboleth has been evolved by Internet2/MACE. It provides a method of distributed authentication and authorization for participating HTTP(S) based applications.	Security Assertion Markup Language is an XML-based open standard data format for exchanging authentication and authorization data between parties. A schema example can be found on OASIS (Advancing open standards for the information society)

Figure B.1: Metamodel standards

Class	Category	Subcategory	Examples
Language	Encoding	Ingest Encoding Mismatch	For example, ASCII *v* UTF-8
		Ingest Encoding Lacking	Mis-recognition of tokens because not being parsed with the proper encoding
		Query Encoding Mismatch	For example, ASCII *v* UTF-8 in search
		Query Encoding Lacking	Mis-recognition of search tokens because not being parsed with the proper encoding
	Languages	Script Mismatch	Variations in how parsers handle, say, stemming, white spaces or hyphens
		Parsing / Morphological Analysis Errors (many)	Arabic languages (right-to-left) *v* Romance languages (left-to-right)
		Syntactical Errors (many)	Ambiguous sentence references, such as *I'm glad I'm a man, and so is Lola* (Lola by Ray Davies and the Kinks)
		Semantics Errors (many)	River *bank v* money *bank v* billiards *bank* shot
Conceptual	Naming	Case Sensitivity	Uppercase *v* lower case *v* Camel case
		Synonyms	United States *v* USA *v* America *v* Uncle Sam *v* Great Satan
		Acronyms	United States *v* USA *v* US
		Homonyms	Such as when the same name refers to more than one concept, such as Name referring to a person *v* Name referring to a book
		Misspellings	As stated
	Generalization / Specialization		When single items in one schema are related to multiple items in another schema, or vice versa. For example, one schema may refer to "phone" but the other schema has multiple elements such as "home phone", "work phone" and "cell phone"
	Aggregation	Intra-aggregation	When the same population is divided differently (such as, Census *v* Federal regions for states, England *v* Great Britain *v* United Kingdom, or full person names *v* first-middle-last)
		Inter-aggregation	May occur when sums or counts are included as set members
	Internal Path Discrepancy		Can arise from different source-target retrieval paths in two different schemas (for example, hierarchical structures where the elements are different levels of remove)
	Missing Item	Content Discrepancy	Differences in set enumerations or including items or not (say, US territories) in a listing of US states
		Missing Content	Differences in scope coverage between two or more datasets for the same concept
		Attribute List Discrepancy	Differences in attribute completeness between two or more datasets
		Missing Attribute	Differences in scope coverage between two or more datasets for the same attribute
	Item Equivalence		When two types (classes or sets) are asserted as being the same when the scope and reference are not (for example, Berlin the city *v* Berlin the official city-state)
			When two individuals are asserted as being the same when they are actually distinct (for example, John F. Kennedy the president *v John F. Kennedy* the aircraft carrier)
	Type Mismatch		When the same item is characterized by different types, such as a person being typed as an animal *v* human being *v* person
	Constraint Mismatch		When attributes referring to the same thing have different cardinalities or disjointedness assertions
		Element-value to Element-	

<table>
<tr><td rowspan="12">Domain</td><td rowspan="5">Schematic Discrepancy</td><td>label Mapping</td><td rowspan="5">One of four errors that may occur when attribute names (say, Hair v Fur) may refer to the same attribute, or when same attribute names (say, Hair v Hair) may refer to different attribute scopes (say, Hair v Fur) or where values for these attributes may be the same but refer to different actual attributes or where values may differ but be for the same attribute and putative value.

Many of the other semantic heterogeneities herein also contribute to schema discrepancies</td></tr>
<tr><td>Attribute-value to Element-label Mapping</td></tr>
<tr><td>Element-value to Attribute-label Mapping</td></tr>
<tr><td>Attribute-value to Attribute-label Mapping</td></tr>
<tr></tr>
<tr><td rowspan="2">Scale or Units</td><td>Measurement Type</td><td>Differences, say, in the metric v English measurement systems, or currencies</td></tr>
<tr><td>Units</td><td>Differences, say, in meters v centimeters v millimeters</td></tr>
<tr><td colspan="2">Precision</td><td>For example, a value of 4.1 inches in one dataset v 4.106 in another dataset</td></tr>
<tr><td rowspan="2">Data representation</td><td>Primitive Data Type</td><td>Confusion often arises in the use of literals v URIs v object types</td></tr>
<tr><td>Data Format</td><td>Delimiting decimals by period v commas; various date formats; using exponents or aggregate units (such as thousands or millions)</td></tr>
<tr></tr>
<tr></tr>
<tr><td rowspan="8">Data</td><td rowspan="5">Naming</td><td>Case Sensitivity</td><td>Uppercase v lower case v Camel case</td></tr>
<tr><td>Synonyms</td><td>For example, centimeters v cm</td></tr>
<tr><td>Acronyms</td><td>For example, currency symbols v currency names</td></tr>
<tr><td>Homonyms</td><td>Such as when the same name refers to more than one attribute, such as Name referring to a person v Name referring to a book</td></tr>
<tr><td>Misspellings</td><td>As stated</td></tr>
<tr><td colspan="2">ID Mismatch or Missing ID</td><td>URIs can be a particular problem here, due to actual mismatches but also use of name spaces or not and truncated URIs</td></tr>
<tr><td colspan="2">Missing Data</td><td>A common problem, more acute with closed world approaches than with open world ones</td></tr>
<tr><td colspan="2">Element Ordering</td><td>Set members can be ordered or unordered, and if ordered, the sequences of individual members or values can differ</td></tr>
</table>

Figure B.2: Classification of Semantic Heterogeneity Sources (See [203])

Listing B.1: OWL/XML Syntax for Ontology

```
<!DOCTYPE Ontology [
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
]>

 <Ontology
   xml:base="http://example.com/owl/families/"
   ontologyIRI="http://example.com/owl/families"
   xmlns="http://www.w3.org/2002/07/owl#">

   <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
   <Prefix name="otherOnt" IRI="http://example.org/otherOntologies/families/"/>
   <Import>http://example.org/otherOntologies/families.owl</Import>

   <!-- Indyviduals -->
   <Declaration>
     <NamedIndividual IRI="John"/>
   </Declaration>
   <Declaration>
     <NamedIndividual IRI="James"/>
   </Declaration>
   <Declaration>
     <NamedIndividual IRI="Jim"/>
   </Declaration>
   <Declaration>
     <NamedIndividual IRI="Mary"/>
   </Declaration>

   <!--Classes -->
   <Declaration>
     <Class IRI="Person"/>
   </Declaration>
   <Declaration>
     <Class IRI="Woman"/>
   </Declaration>
   <Declaration>
     <Class IRI="Parent"/>
   </Declaration>
   <Declaration>
     <Class IRI="Father"/>
   </Declaration>

   <!-- Properties -->
   <Declaration>
     <ObjectProperty IRI="hasWife"/>
   </Declaration>
   <Declaration>
     <ObjectProperty IRI="hasChild"/>
   </Declaration>

   <Declaration>
     <Datatype IRI="personAge"/>
   </Declaration>

    <DatatypeDefinition>
     <Datatype IRI="personAge"/>
     <DatatypeRestriction>
       <Datatype IRI="&xsd;integer"/>
       <FacetRestriction facet="&xsd;minInclusive">
         <Literal datatypeIRI="&xsd;integer">0</Literal>
       </FacetRestriction>
       <FacetRestriction facet="&xsd;maxInclusive">
```

```
        <Literal datatypeIRI="&xsd;integer">150</Literal>
      </FacetRestriction>
    </DatatypeRestriction>
  </DatatypeDefinition>

  <!-- Axiom with Annotation in Class Hierarchy relationships-->
  <SubClassOf>
    <Annotation>
      <AnnotationProperty IRI="&rdfs;comment"/>
      <Literal datatypeIRI="xsd:string">"States that every man is a person."</Literal>
    </Annotation>
    <Class IRI="Man"/>
    <Class IRI="Person"/>
  </SubClassOf>

  <!-- Restriction -->
  <EquivalentClasses>
   <Class IRI="Parent"/>
   <ObjectSomeValuesFrom>
     <ObjectProperty IRI="hasChild"/>
     <Class IRI="Person"/>
   </ObjectSomeValuesFrom>
  </EquivalentClasses>

  <DataPropertyDomain>
    <DataProperty IRI="hasAge"/>
    <Class IRI="Person"/>
  </DataPropertyDomain>
  <DataPropertyRange>
    <DataProperty IRI="hasAge"/>
    <Datatype IRI="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/>
  </DataPropertyRange>

  <SubClassOf>
    <Class IRI="Father"/>
    <ObjectIntersectionOf>
      <Class IRI="Man"/>
      <Class IRI="Parent"/>
    </ObjectIntersectionOf>
  </SubClassOf>

  <SubClassOf>
    <ObjectIntersectionOf>
      <ObjectOneOf>
        <NamedIndividual IRI="Mary"/>
        <NamedIndividual IRI="Bill"/>
        <NamedIndividual IRI="Meg"/>
      </ObjectOneOf>
      <Class IRI="Female"/>
    </ObjectIntersectionOf>
    <ObjectIntersectionOf>
      <Class IRI="Parent"/>
      <ObjectMaxCardinality cardinality="1">
        <ObjectProperty IRI="hasChild"/>
      </ObjectMaxCardinality>
      <ObjectAllValuesFrom>
        <ObjectProperty IRI="hasChild"/>
        <Class IRI="Female"/>
      </ObjectAllValuesFrom>
    </ObjectIntersectionOf>
  </SubClassOf>

  <SameIndividual>
```

```
126      <NamedIndividual IRI="James"/>
127      <NamedIndividual IRI="Jim"/>
128    </SameIndividual>
129
130    <!-- Object Property -->
131    <ObjectPropertyAssertion>
132      <ObjectProperty IRI="hasWife"/>
133      <NamedIndividual IRI="John"/>
134      <NamedIndividual IRI="Mary"/>
135    </ObjectPropertyAssertion>
136
137    <ClassAssertion>
138      <Class IRI="SocialRole"/>
139      <NamedIndividual IRI="Father"/>
140    </ClassAssertion>
141
142  </Ontology>
```

APPENDIX C

Hadoop Ecosystem

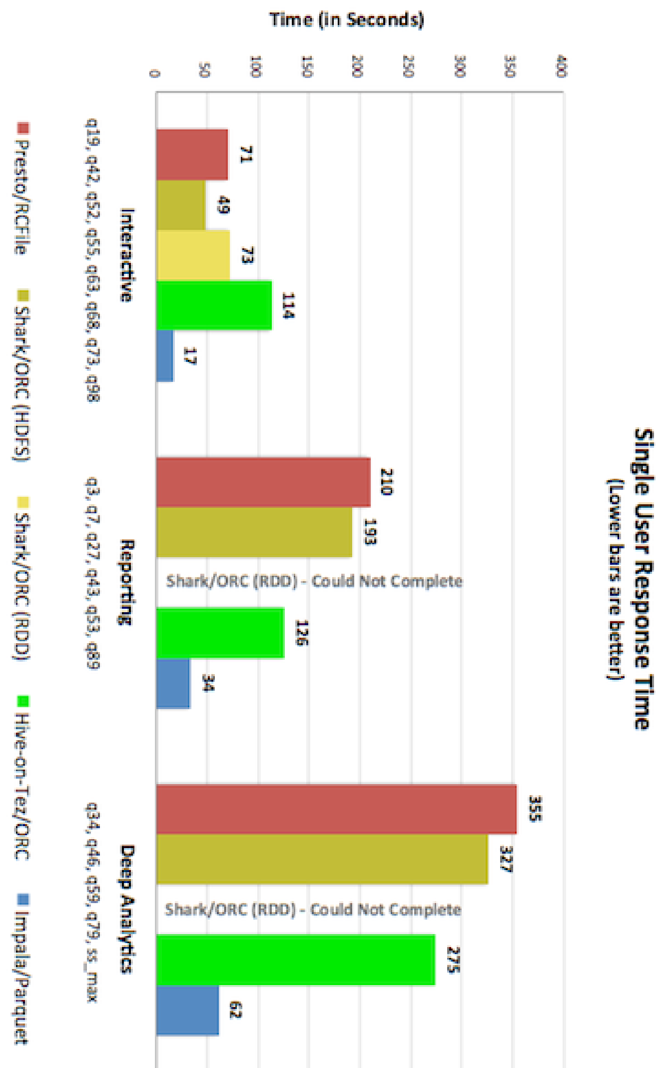


Figure C.1: Single user [1].

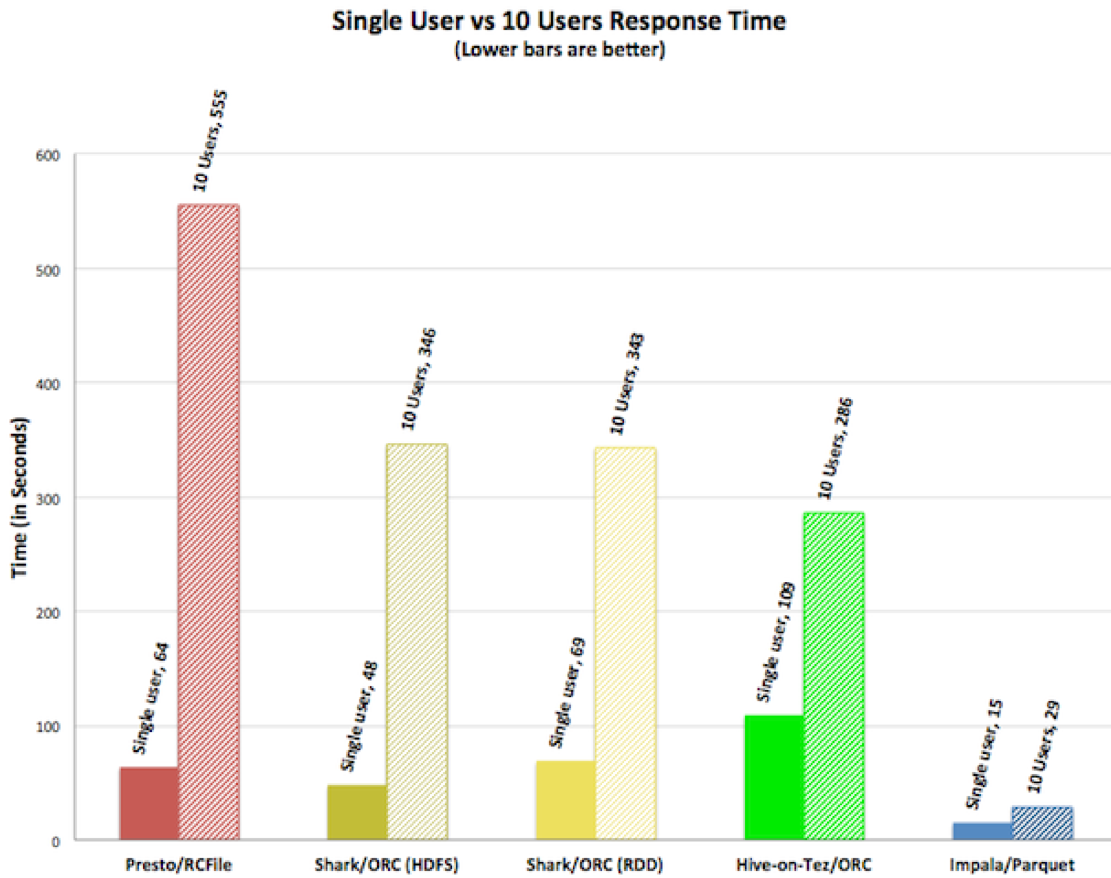


Figure C.2: Multi user.

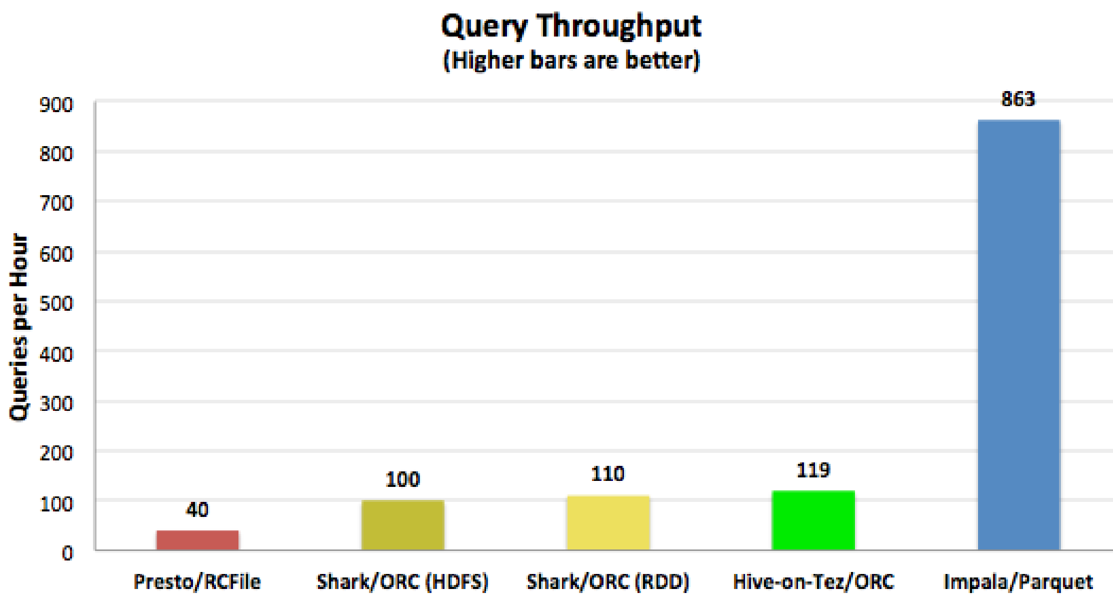


Figure C.3: Query throughput.

[1]Source: Cloudera Inc. http://blog.cloudera.com/wp-content/uploads/2014/05/single-user.png
[2]Source: Hortonworks Inc. http://hortonworks.com/wp-content/uploads/2013/10/Hive12deux.png

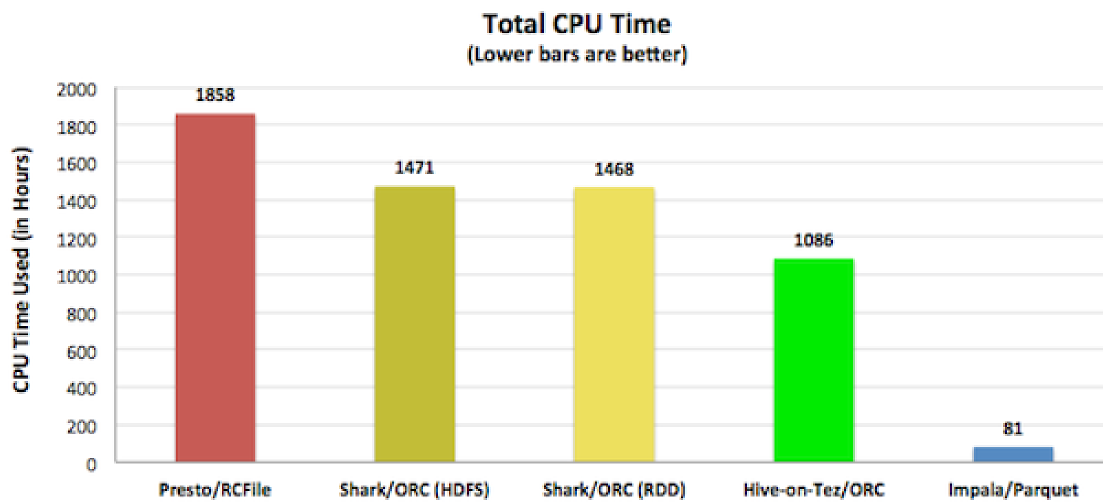


Figure C.4: CPU

Batch Era			
Release	**Date**	**Major Feature(s)**	**Lines of Code**
0.3.0	4/30/09	Initial Release	75600
0.4.0	10/12/09	Map-side Joins	170100
0.5.0	2/23/10	Multi File Format	77100
0.6.0	10/29/10	Sorted Merge Join	130300
0.7.0	3/29/11	Indexing	111800
0.8.0	12/16/11	Indexed GROUP BY	153500
0.9.0	4/30/12	NULL-safe equijoins	62800
0.10.0	1/11/13	Skew JOIN	147400

(a) Hive 0.10

Interactive Era			
Release	**Date**	**Major Feature(s)**	**Lines of Code**
0.11.0	5/15/13	ORCFile, OLAP Functions	132000
0.12.0	10/15/13	SQL Data Types	94100
0.13.0	4/21/14	Hive on Tez, Vectorized Query	419300
0.14.0	11/12/14	CBO, ACID	340500
1.0	**2/5/15**	**STABLE FOUNDATION**	

(b) Hive 0.13

Sub Second Era			
Release	**Date**	**Major Feature(s)**	**Lines of Code**
1.1.0+	-	LLAP, Hive on Spark. etc	

(c) Hive / Stinger.next

Figure C.5: Hive Stinger Evolution

Notable Improvements in Hive 12

Speed
- 10x Faster Query Launch for Databases w/ 500+ Partitions
- Faster Query Plan Generation
- ORCFile Predicate Pushdown
- Parallel ORDER BY
- Correlation Optimizer
- Push LIMIT down to mappers
- Optimize GROUP BY followed by JOIN
- Optimizations to COUNT

SQL
- VARCHAR Support
- DATE Support
- Support for Macros Within Hive Queries
- GROUP BY on structs and unions
- Lateral View Improvements: Outer Lateral Views
- Truncation of columns (or tables)

Other
- HCatalog integrated into Hive
- Support for Encryption Between Hive and BI Tools
- Improved HBase Integration
- Ability to Move Partitions Between Tables
- Partition pruning info in explain plans

(a) Hive 0.12

Notable Improvements In Apache Hive 0.13

Speed
- Interactive query through Hive on Tez
- Vectorized query execution engine
- Cost-based optimizer
- Split elimination for ORCFile
- Partition pruning for string and date datatypes
- Faster query planning

Scale
- More scalable dynamic partition loads
- Smaller hash tables, allowing more scalable MapJoins

SQL
- Subquery for IN / NOT IN
- Support for EXISTS and NOT EXISTS
- Common table expressions (CTEs)
- Support for CHAR datatype
- Scale and precision for DECIMAL datatype
- Authorization based on SQL standards
- JOIN conditions in the WHERE clause
- Support for non-ASCII column names
- Permanent UDFs
- Stream data into Hive from Flume (Experimental feature)

Additional Features
- HiveServer 2 improvements
  - PAM authentication support
  - SSL encryption
  - HTTP/HTTPS support
  - Cancel MR/Tez jobs via JDBC/ODBC
- HCatalog parity for all of Hive datatypes

(b) Hive 0.13

Notable Improvements in Apache Hive 0.14
- Transactions with ACID semantics
- Cost based optimizer
- SQL temporary tables
- Apache Accumulo integration

(c) Hive 0.14

Figure C.6: Hive Stinger Phases [2]

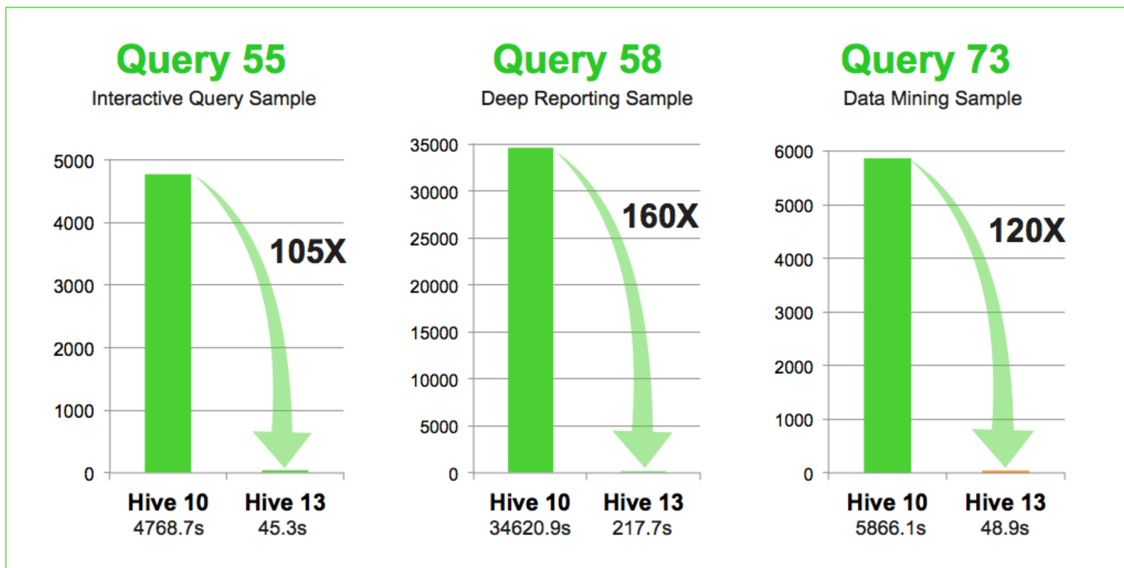


Figure C.7: Hive versions benchmarks for Stinger Initiative results

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 19, 118).

[2] Gio Wiederhold. "Mediators in the Architecture of Future Information Systems". In: *Computer* 25.3 (Mar. 1992), pp. 38–49. DOI: 10.1109/2.121508 (cit. on p. 22).

[3] Rafi Ahmed, Joseph Albert, Weimin Du, William Kent, Witold Litwin, and Ming-Chien Shan. "An Overview of Pegasus". In: *RIDE-IMS '93, Thirst International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria, April 19-20, 1993.* 1993, pp. 273–277. DOI: 10.1109/RIDE.1993.281908 (cit. on p. 22).

[4] Joseph Albert, Rafi Ahmed, Mohammad A. Ketabchi, William Kent, and Ming-Chien Shan. "Automatic Importation of Relational Schemas in Pegasus". In: *RIDE-IMS '93, Thirst International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria, April 19-20, 1993.* 1993, pp. 105–113. DOI: 10.1109/RIDE.1993.281938 (cit. on p. 22).

[5] Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, and William Kent. "Pegasus: A Heterogeneous Information Management System". In: *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Ed. by Won Kim. ACM Press and Addison-Wesley, 1995, pp. 664–682 (cit. on p. 22).

[6] Gustav Fahl, Tore Risch, and Martin Sköld. "AMOS - An Architecture for Active Mediators". In: *NGITS' 93 - The International Workshop on Next Generation Information Technologies and Systems, Technion, Haifa, Israel, June 28-30, 1993.* Ed. by Opher Etzion and Arie Segev. 1993, pp. 47–53 (cit. on p. 22).

[7] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. "Scaling Access to Heterogeneous Data Sources with DISCO". In: *IEEE Trans. Knowl. Data Eng.* 10.5 (1998), pp. 808–823. DOI: 10.1109/69.729736 (cit. on p. 22).

[8] *Oracle Data Integrator*. URL: http://www-03.ibm.com/software/products/pl/infoinfoservfordatainte (visited on 08/01/2015) (cit. on p. 22).

[9] *InfoSphere Information Server for Data Integration.* (Visited on 08/01/2015) (cit. on p. 22).

[10] *Oracle GoldenGate*. URL: http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html (visited on 08/01/2015) (cit. on p. 22).

[11] *Pentaho Data Integration.* URL: http://www.pentaho.com/product/data-integration (visited on 08/01/2015) (cit. on p. 22).

[12] *Talend Data Integration.* URL: https://www.talend.com/products/data-integration (visited on 08/01/2015) (cit. on p. 22).

[13] Edgar F Codd. *Relational completeness of data base sublanguages.* IBM Corporation, 1972 (cit. on pp. 23, 52).

[14] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. "The TSIMMIS Project: Integration of Heterogeneous Information Sources". In: *IPSJ.* 1994, pp. 7–18 (cit. on p. 23).

[15] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas II, John H. Williams, and Edward L. Wimmers. "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach". In: *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, March 6-7, 1995.* Ed. by Omran A. Bukhres, M. Tamer Özsu, and Ming-Chien Shan. IEEE Computer Society, 1995, pp. 124–131. DOI: 10.1109/RIDE.1995.378736 (cit. on p. 23).

[16] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. "Mariposa: A Wide-Area Distributed Database System". In: *VLDB J.* 5.1 (1996), pp. 48–63 (cit. on p. 23).

[17] C. Batini, M. Lenzerini, and S. B. Navathe. "A Comparative Analysis of Methodologies for Database Schema Integration". In: *ACM Comput. Surv.* 18.4 (Dec. 1986), pp. 323–364. DOI: 10.1145/27633.27634 (cit. on p. 23).

[18] Maurizio Lenzerini. "Data Integration: A Theoretical Perspective". In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems.* PODS '02. Madison, Wisconsin: ACM, 2002, pp. 233–246. DOI: 10.1145/543613.543644 (cit. on pp. 23, 39).

[19] Richard Hull. "Managing Semantic Heterogeneity in Databases: A Theoretical Prospective". In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* PODS '97. Tucson, Arizona, USA: ACM, 1997, pp. 51–61. DOI: 10.1145/263661.263668 (cit. on p. 23).

[20] Michael Franklin, Alon Halevy, and David Maier. "From Databases to Dataspaces: A New Abstraction for Information Management". In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 27–33. DOI: 10.1145/1107499.1107502 (cit. on p. 23).

[21] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, Dave Maier, Timothy Mattson, and Stan Zdonik. "The BigDAWG Polystore System". In: *ACM Sigmod Record* 44.3 (2015) (cit. on pp. 23, 145, 146).

[22] *Apache Hive ACID Transactions in HDP 2.2.* URL: http://hortonworks.com/blog/apache-hive-acid-transactions-hdp-2-2/ (visited on 08/01/2015) (cit. on p. 23).

[23] Theo Haerder and Andreas Reuter. "Principles of Transaction-oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. DOI: 10.1145/289.291 (cit. on pp. 26, 28).

[24] W. C. Mcgee. "Data Base Technology". In: *ZBM J. Res. Develop* 25 (1981), pp. 505–519 (cit. on p. 28).

[25] E. F. Codd. "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks". In: *IBM Research Report, San Jose, California* RJ599 (1969) (cit. on pp. 28, 54).

[26] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. DOI: 10.1145/362384.362685 (cit. on pp. 28, 51, 52).

[27] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’79. Boston, Massachusetts: ACM, 1979, pp. 23–34. DOI: 10.1145/582095.582099 (cit. on pp. 28, 51, 147).

[28] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. “System R: Relational Approach to Database Management”. In: *ACM Trans. Database Syst.* 1.2 (June 1976), pp. 97–137. DOI: 10.1145/320455.320457 (cit. on pp. 28, 51).

[29] Don Chamberlin. *Bibliography of the System R Project*. Updated 4 December 2000 by Paul McJones. Sept. 26, 1995. URL: http://www.mcjones.org/System_R/bib.html (visited on 02/26/2015) (cit. on pp. 28, 51).

[30] Jim Gray. “The Transaction Concept: Virtues and Limitations (Invited Paper)”. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB ’81. Cannes, France: VLDB Endowment, 1981, pp. 144–154 (cit. on p. 28).

[31] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’74. Ann Arbor, Michigan: ACM, 1974, pp. 249–264. DOI: 10.1145/800296.811515 (cit. on p. 29).

[32] André B. Bondi. “Characteristics of Scalability and Their Impact on Performance”. In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP ’00. Ottawa, Ontario, Canada: ACM, 2000, pp. 195–203. DOI: 10.1145/350391.350432 (cit. on p. 30).

[33] Mark D. Hill. “What is Scalability?” In: *SIGARCH Comput. Archit. News* 18.4 (Dec. 1990), pp. 18–21. DOI: 10.1145/121973.121975 (cit. on p. 30).

[34] Peter Bailis and Ali Ghodsi. “Eventual Consistency Today: Limitations, Extensions, and Beyond”. In: *Queue* 11.3 (Mar. 2013), 20:20–20:32. DOI: 10.1145/2460276.2462076 (cit. on p. 32).

[35] *Hadoop Deployment Comparison Study*. 2013. URL: http://www.accenture.com/SiteCollectionDocuments/PDF/Accenture-Hadoop-Deployment-Comparison-Study.pdf (visited on 03/03/2015) (cit. on p. 33).

[36] Raghunath Nambiar and Meikel Poess, eds. *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers*. Vol. 8904. Lecture Notes in Computer Science. Springer, 2015. DOI: 10.1007/978-3-319-15350-6 (cit. on p. 34).

[37] Mike Lang. *Making Hadoop suitable for enterprise data science*. Tech. rep. Technology Forecast: Rethinking integration, Issue 1, 2014. PwC’s Center for Technology and Innovation (CTI), 2014 (cit. on pp. 34, 84).

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1626–1629. DOI: 10.14778/1687553.1687609 (cit. on p. 34).

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. "Hive - a petabyte scale data warehouse using Hadoop". In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. Ed. by Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras. IEEE, 2010, pp. 996–1005. DOI: 10.1109/ICDE.2010.5447738 (cit. on p. 34).

[40] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. "Data Warehousing and Analytics Infrastructure at Facebook". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1013–1020. DOI: 10.1145/1807167.1807278 (cit. on p. 34).

[41] Matthew Aslett - 451 Group. *How Will The Database Incumbents Respond To NoSQL And NewSQL?* Apr. 4, 2011. URL: http://cs.brown.edu/courses/cs227/archives/2012/papers/newsql/aslett-newsql.pdf (visited on 02/26/2015) (cit. on p. 34).

[42] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong. "Multibase: Integrating Heterogeneous Distributed Database Systems". In: *Proceedings of the May 4-7, 1981, National Computer Conference*. AFIPS '81. Chicago, Illinois: ACM, 1981, pp. 487–499. DOI: 10.1145/1500412.1500483 (cit. on p. 38).

[43] Michal Chromiak and Marcin Grabowiecki. "Heterogeneous Data Integration Architecture-Challenging Integration Issues". In: *Annales UMCS, Informatica* 15.1 (2015) (cit. on p. 38).

[44] Dennis Heimbigner and Dennis McLeod. "A Federated Architecture for Information Management". In: *ACM Trans. Inf. Syst.* 3.3 (July 1985), pp. 253–278. DOI: 10.1145/4229.4233 (cit. on p. 39).

[45] Ling Liu and M. Tamer Zsu. *Encyclopedia of Database Systems*. 1st. Springer Publishing Company, Incorporated, 2009 (cit. on p. 40).

[46] Thomas R. Gruber. "A Translation Approach to Portable Ontology Specifications". In: *Knowl. Acquis.* 5.2 (June 1993), pp. 199–220. DOI: 10.1006/knac.1993.1008 (cit. on p. 40).

[47] Thomas R. Gruber. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing". In: *Int. J. Hum.-Comput. Stud.* 43.5-6 (Dec. 1995), pp. 907–928. DOI: 10.1006/ijhc.1995.1081 (cit. on p. 40).

[48] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. "Ontology-Based Integration of Information - A Survey of Existing Approaches". In: *In IJCAI'01 Workshop. on Ontologies and Information Sharing*. 2001 (cit. on p. 40).

[49] Lorena Otero-Cerdeira, Francisco Javier Rodríguez-Martínez, and Alma Gómez-Rodríguez. "Ontology matching: A literature review". In: *Expert Syst. Appl.* 42.2 (2015), pp. 949–971. DOI: 10.1016/j.eswa.2014.08.032 (cit. on p. 41).

[50] The ARTFL Project. *Webster's Revised Unabridged Dictionary (1913 + 1828)*. URL: http://machaut.uchicago.edu/?action=search&word=heterogeneous&resource=Webster's&quicksearch=on (visited on 05/02/2015) (cit. on p. 47).

[51] William Kent. "The Many Forms of a Single Fact". In: *Proc. IEEE COMPCON*. COMPCON 1989. San Francisco, USA: Also Hewlett-Packard Laboratories (HPL-SAL-88-8), 1989 (cit. on p. 48).

[52] Charnyote Pluempitiwiriyawej and Joachim Hammer. *A Classification Scheme for Semantic and Schematic Heterogeneities in XML Data Sources.* Tech. rep. TR00-004. University of Florida, Sept. 2000 (cit. on p. 48).

[53] Mike Bergman. *Sources and Classification of Semantic Heterogeneities.* June 6, 2006. URL: `http://www.mkbergman.com/232/sources-and-classification-of-semantic-heterogeneities/` (visited on 05/02/2015) (cit. on p. 49).

[54] W3C Recommendation 11 February 2014. *Efficient XML Interchange (EXI) Format 1.0 (Second Edition).* Feb. 11, 2014. URL: `http://www.w3.org/TR/exi/` (visited on 05/13/2015) (cit. on p. 51).

[55] Michel Lacroix and Alain Pirotte. "Domain-oriented Relational Languages". In: *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3.* VLDB '77. Tokyo, Japan: VLDB Endowment, 1977, pp. 370–378 (cit. on p. 52).

[56] Matthias Jarke and Jurgen Koch. "Query Optimization in Database Systems". In: *ACM Comput. Surv.* 16.2 (June 1984), pp. 111–152. DOI: `10.1145/356924.356928` (cit. on p. 52).

[57] Hugh Darwen and C. J. Date. "The Third Manifesto". In: *SIGMOD Rec.* 24.1 (Mar. 1995), pp. 39–49. DOI: `10.1145/202660.202667` (cit. on p. 52).

[58] E. F. Codd. "Further Normalization of the Data Base Relational Model". In: *IBM Research Report, San Jose, California* RJ909 (1971) (cit. on p. 54).

[59] Edgar F. Codd. *Recent Investigations into Relational Data Base Systems.* Tech. rep. RJ1385. IBM, Apr. 1974 (cit. on p. 54).

[60] Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features.* New York, NY, USA: Elsevier Science Inc., 2002 (cit. on p. 54).

[61] Kazimierz Subieta. *Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL).* July 2, 2011. URL: `http://sbql.pl` (visited on 05/17/2015) (cit. on p. 56).

[62] Krzysztof Stencel. *LoXiM - an experimental Database Management System.* Mar. 13, 2009. URL: `http://loxim.sourceforge.net/` (visited on 05/17/2015) (cit. on p. 56).

[63] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. "Column-oriented Database Systems". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1664–1665. DOI: `10.14778/1687553.1687625` (cit. on p. 57).

[64] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. "Column-stores vs. Row-stores: How Different Are They Really?" In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 967–980. DOI: `10.1145/1376616.1376712` (cit. on p. 57).

[65] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. "C-store: A Column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases.* VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564 (cit. on p. 60).

[66] W3C. *RDF/XML Syntax Specification (Revised), W3C Recommendation.* Feb. 10, 2004. URL: `http://www.w3.org/TR/REC-rdf-syntax/` (visited on 03/02/2015) (cit. on p. 62).

[67] Gioldasis N. Bikakis N. Tsinaraki C., Stavrakantonakis I., and Christodoulakis S. *The XML and Semantic Web Worlds: Technologies, Interoperability and Integration. A survey of the State of the Art.* 2013. URL: `http://www.dblab.ntua.gr/~bikakis/XMLSemanticWebW3CTimeline.pdf` (visited on 03/02/2015) (cit. on p. 62).

[68] Stephen Pimentel. *The rise of the multimodel database.* Jan. 6, 2015. URL: http://www.infoworld.com/article/2861579/database/the-rise-of-the-multimodel-database.html (visited on 03/02/2015) (cit. on p. 63).

[69] Michael Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps.* Answering a comment as a signed CACM Administrator. June 22, 2011. URL: http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext (visited on 05/17/2015) (cit. on p. 64).

[70] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP Through the Looking Glass, and What We Found There". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 981–992. DOI: 10.1145/1376616.1376713 (cit. on p. 64).

[71] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. "Anti-caching: A New Approach to Database Management System Architecture". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1942–1953. DOI: 10.14778/2556549.2556575 (cit. on p. 64).

[72] Jun Rao and Kenneth A. Ross. "Making B+- Trees Cache Conscious in Main Memory". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 475–486. DOI: 10.1145/335191.335449 (cit. on p. 65).

[73] Burkhard Stiller, Thomas Bocek, Fabio Hecht, Guilherme Machado, Peter Racz, and Martin Waldburger. *TECHNICAL OVERVIEW High performance, scalable RDBMS for Big Data and Real-time Analytics.* Tech. rep. VoltDB Inc, May 2012 (cit. on p. 65).

[74] Martin Hilbert and Priscila López. "The world's technological capacity to store, communicate, and compute information". In: *science* 332.6025 (2011), pp. 60–65 (cit. on p. 66).

[75] IBM Inc. *What is big data?* 2013. URL: http://www-01.ibm.com/software/au/data/bigdata/ (visited on 03/02/2015) (cit. on p. 66).

[76] Cisco Inc. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2014–2019 White Paper.* Feb. 3, 2015. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html (visited on 03/02/2015) (cit. on p. 66).

[77] Ericsson Inc. *Ericsson Mobility Report On the Pulse of the Networked Society.* June 2014. URL: http://www.ericsson.com/res/docs/2014/ericsson-mobility-report-june-2014.pdf (visited on 03/02/2015) (cit. on p. 67).

[78] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. "Big data: Big gaps of knowledge in the field of internet science". In: *International Journal of Internet Science* 7.1 (2012), pp. 1–5 (cit. on p. 67).

[79] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. "The rise of "big data" on cloud computing: Review and open research issues". In: *Inf. Syst.* 47 (2015), pp. 98–115. DOI: 10.1016/j.is.2014.07.006 (cit. on p. 67).

[80] Douglas Laney. *3D Data Management: Controlling Data Volume, Velocity, and Variety.* Tech. rep. META Group, 2001 (cit. on p. 67).

[81] *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data.* STAMFORD, Conn. June 27, 2011. URL: http://www.gartner.com/newsroom/id/1731916 (visited on 04/16/2015) (cit. on p. 67).

[82] Douglas Laney Mark A. Beyer. *The Importance of 'Big Data': A Definition*. June 21, 2012. URL: https://www.gartner.com/doc/2057415/importance-big-data-definition (visited on 04/16/2015) (cit. on p. 73).

[83] Andrea De Mauro, Marco Greco, and Michele Grimaldi. "What is Big Data? A Consensual Definition and a Review of Key Research Topics". In: *4th International Conference on Integrated Information, Madrid. doi*. Vol. 10. 2.1. 2014, pp. 2341–5048 (cit. on p. 68).

[84] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. "CRDTs: Consistency without concurrency control". In: *CoRR* abs/0907.0929 (2009) (cit. on p. 70).

[85] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400 (cit. on p. 70).

[86] Matt Aslett. *Big data reconsidered: it's the economics, stupid*. Dec. 2, 2013. URL: https://451research.com/report-short?entityId=79479&referrer=marketing (visited on 05/18/2015) (cit. on p. 71).

[87] OMG - Object Management Group. *OMG:Catalog of OMG CORBA®/IIOP® Specifications*. 2012. URL: http://www.omg.org/spec/CORBA/Current/ (visited on 02/01/2015) (cit. on pp. 71, 99, 115, 117).

[88] Michał Chromiak, Krzysztof Stencel, and Kazimierz Subieta. "Indexing Distributed and Heterogeneous Resources". In: *U- and E-Service, Science and Technology - International Conference UNESST 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*. Ed. by Tai-Hoon Kim, Jianhua Ma, Wai-Chi Fang, Byungjoo Park, Byeong Ho Kang, and Dominik Slezak. Vol. 124. Communications in Computer and Information Science. Springer, 2010, pp. 214–223. DOI: 10.1007/978-3-642-17644-9_24 (cit. on p. 71).

[89] Tomasz Marek Kowalski, Michał Chromiak, Kamil Kuliberda, Jacek Wislicki, Radoslaw Adamus, and Kazimierz Subieta. "Query Optimization by Indexing in the ODRA OODBMS". In: *Annales UMCS, Informatica* 9.1 (2009), pp. 77–97. DOI: 10.2478/v10065-009-0006-z (cit. on p. 71).

[90] Michał Chromiak, Piotr Wisniewski, and Krzysztof Stencel. "Exploiting Order Dependencies on Primary Keys for Optimization". In: *Proceedings of the 23th International Workshop on Concurrency, Specification and Programming, Chemnitz, Germany, September 29 - October 1, 2014*. Ed. by Louchka Popova-Zeugmann. Vol. 1269. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 58–68 (cit. on pp. 72, 157, 161).

[91] E.F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate*. Codd & Associates, 1993 (cit. on p. 72).

[92] Ee-Peng Lim, Hsinchun Chen, and Guoqing Chen. "Business Intelligence and Analytics: Research Directions". In: *ACM Trans. Manage. Inf. Syst.* 3.4 (Jan. 2013), 17:1–17:10. DOI: 10.1145/2407740.2407741 (cit. on p. 73).

[93] Daniel J. Power and Ramesh Sharda. "Model-driven Decision Support Systems: Concepts and Research Directions". In: *Decis. Support Syst.* 43.3 (Apr. 2007), pp. 1044–1061. DOI: 10.1016/j.dss.2005.05.030 (cit. on p. 74).

[94] Stephen Haag. *Management Information Systems for the Information Age*. McGraw-Hill Higher Education, 2000 (cit. on p. 74).

[95] P. Ponniah. *Data Warehousing Fundamentals for IT Professionals*. Wiley, 2011 (cit. on p. 74).

[96] Yingying Tao, Qiang Zhu, Calisto Zuzarte, and Wing Lau. “Optimizing Large Star-schema Queries with Snowflakes via Heuristic-based Query Rewriting”. In: *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research.* CASCON ’03. Toronto, Ontario, Canada: IBM Press, 2003, pp. 279–293 (cit. on p. 75).

[97] Morteza Zaker, Somnuk Phon-Amnuaisuk, and Su-Cheng Haw. “Investigating Design Choices Between Bitmap Index and B-tree Index for a Large Data Warehouse System”. In: *Proceedings of the 8th Conference on Applied Computer Scince.* ACS’08. Venice, Italy: World Scientific, Engineering Academy, and Society (WSEAS), 2008, pp. 123–130 (cit. on p. 75).

[98] Patrick O’Neil and Dallan Quass. “Improved Query Performance with Variant Indexes”. In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’97. Tucson, Arizona, USA: ACM, 1997, pp. 38–49. DOI: 10.1145/253260.253268 (cit. on p. 75).

[99] Kamesh Madduri and Kesheng Wu. “Efficient Joins with Compressed Bitmap Indexes”. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management.* CIKM ’09. Hong Kong, China: ACM, 2009, pp. 1017–1026. DOI: 10.1145/1645953.1646083 (cit. on p. 75).

[100] Theodore Johnson. “Performance Measurements of Compressed Bitmap Indices”. In: *Proceedings of the 25th International Conference on Very Large Data Bases.* VLDB ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–289 (cit. on p. 75).

[101] Sebastiaan J. van Schaik and Oege de Moor. “A Memory Efficient Reachability Data Structure Through Bit Vector Compression”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’11. Athens, Greece: ACM, 2011, pp. 913–924. DOI: 10.1145/1989323.1989419 (cit. on p. 75).

[102] Deepak Agarwal, Datong Chen, Long-ji Lin, Jayavel Shanmugasundaram, and Erik Vee. “Forecasting High-dimensional Data”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1003–1012. DOI: 10.1145/1807167.1807277 (cit. on p. 75).

[103] Ulrike Fischer, Frank Rosenthal, and Wolfgang Lehner. “Sample-based Forecasting Exploiting Hierarchical Time Series”. In: *Proceedings of the 16th International Database Engineering & Applications Sysmposium.* IDEAS ’12. Prague, Czech Republic: ACM, 2012, pp. 120–129. DOI: 10.1145/2351476.2351490 (cit. on p. 75).

[104] W.H. Inmon. *Building the Data Warehouse.* Timely, practical, reliable. Wiley, 2005 (cit. on p. 75).

[105] R. Hillard. *Information-Driven Business: How to Manage Data and Information for Maximum Advantage.* Wiley, 2010 (cit. on p. 75).

[106] Martin Staudt, Anca Vaduva, and Thomas Vetterli. *The Role of Metadata for Data Warehousing.* Tech. rep. 1999 (cit. on p. 76).

[107] Ralph Kimball, Margy Ross, Warren Thornthwaite, Joy Mundy, and Bob Becker. *The Data Warehouse Lifecycle Toolkit.* 2nd. Wiley Publishing, 2008, pp. 10, 115–117, 131–132, 140, 154–155 (cit. on pp. 76, 79, 101).

[108] *CWM 1.1 Specification.* 2007. URL: http://www.cwmforum.org/ (visited on 03/05/2015) (cit. on p. 76).

[109] *Common Warehouse Metamodel, v1.1.* Mar. 2, 2003. URL: http://www.omg.org/spec/CWM/1.1/ (visited on 03/05/2015) (cit. on p. 76).

[110] Torben Bach Pedersen and Christian S. Jensen. “Multidimensional Database Technology”. In: *Computer* 34.12 (Dec. 2001), pp. 40–46. DOI: 10.1109/2.970558 (cit. on p. 77).

[111] Surajit Chaudhuri and Umeshwar Dayal. “An Overview of Data Warehousing and OLAP Technology”. In: *SIGMOD Rec.* 26.1 (Mar. 1997), pp. 65–74. DOI: 10.1145/248603.248616 (cit. on p. 78).

[112] K. Roebuck. *Application Portfolio Management (APM): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors.* Emereo Publishing, 2012 (cit. on p. 78).

[113] P.R. Bagley. *Extension of Programming Language Concepts.* National Bureau of Standards, Institute for Applied Technology, 1968 (cit. on p. 78).

[114] Francis P. Bretherton and Paul T. Singley. “Metadata: A User’s View”. In: *Proceedings of the Seventh International Working Conference on Scientific and Statistical Database Management.* Washington, DC, USA: IEEE Computer Society, 1994, pp. 166–174 (cit. on p. 79).

[115] National Information Standards Organization (U.S.) *Understanding Metadata.* NISO Press, 2004 (cit. on p. 79).

[116] *1484.12.1-2002/Cor 1-2011 - IEEE Standard for Learning Object Metadata - Corrigendum 1: Corrigenda for 1484.12.1 LOM (Learning Object Metadata).* 2011. URL: http://standards.ieee.org/findstds/standard/1484.12.1-2002-Cor_1-2011.html (visited on 03/05/2015) (cit. on p. 79).

[117] *DCMI Specifications.* URL: http://dublincore.org/specifications/ (visited on 03/05/2015) (cit. on p. 79).

[118] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6.* OSDI’04. San Francisco, CA: USENIX Association, 2004, pp. 10–10 (cit. on p. 80).

[119] Message P Forum. *MPI: A Message-Passing Interface Standard.* Tech. rep. Knoxville, TN, USA, 1994 (cit. on p. 80).

[120] Jeffrey D. Ullman. “Designing Good MapReduce Algorithms”. In: *XRDS* 19.1 (Sept. 2012), pp. 30–34. DOI: 10.1145/2331042.2331053 (cit. on p. 82).

[121] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing.* 1st. Berkely, CA, USA: Apress, 2010 (cit. on p. 83).

[122] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. “Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis”. In: *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing.* Science Cloud ’13. New York, New York, USA: ACM, 2013, pp. 13–20. DOI: 10.1145/2465848.2465849 (cit. on p. 83).

[123] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. “A Comparison of Join Algorithms for Log Processing in MaPreduce”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 975–986. DOI: 10.1145/1807167.1807273 (cit. on p. 84).

[124] Oinam Martina Devi Mani Bushan Balaraj J. “Comparison of Join Algorithms in Map Reduce Framework”. In: *International Journal of Innovative Research in Computer and Communication Engineering* Vol.2,Special Issue 5 (Dec. 2014) (cit. on p. 84).

[125] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. "A Comparison of Approaches to Large-scale Data Analysis". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data.* SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 165–178. DOI: 10.1145/1559845.1559865 (cit. on pp. 84, 86, 93).

[126] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proc. of the 36th Int'l Conf on Very Large Data Bases.* 2010, pp. 330–339 (cit. on p. 84).

[127] *Altior's AltraSTAR - Hadoop Storage Accelerator and Optimizer Now Certified on CDH4 (Cloudera's Distribution Including Apache Hadoop Version 4).* Dec. 18, 2012. URL: http://www.prnewswire.com/news-releases/altiors-altrastar---hadoop-storage-accelerator-and-optimizer-now-certified-on-cdh4-clouderas-distribution-including-apache-hadoop-version-4-183906141.html (visited on 03/05/2015) (cit. on p. 84).

[128] Tom White. *Hadoop: The Definitive Guide.* 1st. O'Reilly Media, Inc., 2009 (cit. on p. 85).

[129] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Shark: SQL and Rich Analytics at Scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 13–24. DOI: 10.1145/2463676.2465288 (cit. on p. 86).

[130] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. "MapReduce and Parallel DBMSs: Friends or Foes?" In: *Communications of the ACM* 53.1 (Jan. 2010), pp. 64–71. DOI: 10.1145/1629175.1629197 (cit. on p. 86).

[131] *HBase/PoweredBy.* last edited 2014-08-04 07:11:48 by VitaliyVerbenko. 2014. URL: http://wiki.apache.org/hadoop/Hbase/PoweredBy (visited on 03/03/2015) (cit. on p. 86).

[132] *Cloudera Debuts Real-Time Hadoop Query.* last edited 2012-10-24 11:32 AM by Doug Henschen. 2012. URL: http://www.informationweek.com/software/information-management/cloudera-debuts-real-time-hadoop-query/d/d-id/1107018 (visited on 04/09/2015) (cit. on pp. 86, 87).

[133] *Impala v Hive.* last edited 2013-12-22 11:32 AM by Mike Olson. 2013. URL: http://vision.cloudera.com/impala-v-hive/ (visited on 04/09/2015) (cit. on p. 86).

[134] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181 (cit. on p. 88).

[135] Leslie G. Valiant. "A Bridging Model for Multi-core Computing". In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 154–166. DOI: 10.1016/j.jcss.2010.06.012 (cit. on p. 88).

[136] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. DOI: 10.1145/1807167.1807184 (cit. on p. 88).

[137] *YARN/PoweredBy.* last edited 2011-09-07 17:08:26 by Matei Zaharia. 2014. URL: http://wiki.apache.org/hadoop/PoweredByYarn (visited on 03/03/2015) (cit. on p. 89).

[138] *Hadoop Roadmap.* last edited 2015-03-30 08:38:25 by Akira Ajisaka. 2014. URL: http://wiki.apache.org/hadoop/PoweredByYarn (visited on 03/03/2015) (cit. on p. 90).

[139] *Benchmarking Apache Hive 13 for Enterprise Hadoop.* By Carter Shanklin on June 2nd, 2014. 2014. URL: http://hortonworks.com/blog/benchmarking-apache-hive-13-enterprise-hadoop/ (visited on 03/03/2015) (cit. on p. 91).

[140] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 59–72. DOI: 10.1145/1272998.1273005 (cit. on pp. 91, 93).

[141] *HIVE 0.14 Cost Based Optimizer (CBO) Technical Overview.* March 2nd, 2015. 2015. URL: http://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/ (visited on 04/16/2015) (cit. on p. 92).

[142] *ACID and Transactions in Hive.* last updated by Lefty Leverenz on Apr 13, 2015. 2015. URL: https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions (visited on 04/16/2015) (cit. on p. 93).

[143] *JIRA request: Implement insert, update, and delete in Hive with full ACID support.* last updated 2015-03-12. Started: 2013. URL: https://issues.apache.org/jira/browse/HIVE-5317 (visited on 04/16/2015) (cit. on p. 93).

[144] *Stinger.next: Enterprise SQL at Hadoop Scale with Apache Hive.* September 3rd, 2014. 2014. URL: http://hortonworks.com/blog/stinger-next-enterprise-sql-hadoop-scale-apache-hive/ (visited on 04/16/2015) (cit. on p. 93).

[145] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing.* HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10 (cit. on p. 93).

[146] *Big Data Benchmark.* February 2014. 2014. URL: https://amplab.cs.berkeley.edu/benchmark/ (visited on 04/16/2015) (cit. on p. 93).

[147] *eXtensible Access Control Markup Language (XACML) Version 3.0.* Jan. 22, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (visited on 05/19/2015) (cit. on p. 96).

[148] *Running your SOA like a Web startup.* June 13, 2009. URL: http://www.zdnet.com/article/running-your-soa-like-a-web-startup/ (visited on 05/19/2015) (cit. on p. 97).

[149] *Apache Sqoop.* May 18, 2015. URL: http://sqoop.apache.org/ (visited on 05/19/2015) (cit. on p. 97).

[150] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. OUP USA, 1977 (cit. on p. 100).

[151] *FTP Replacement: Where MFT Makes Sense and Why You Should Care.* Nov. 8, 2010. URL: https://www.gartner.com/doc/1464916 (visited on 06/13/2015) (cit. on p. 101).

[152] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on p. 101).

[153] Julie Gable. *Enterprise Applications - Adoption of E-Business and Document Technologies: 2000-2001 Worldwide.* Tech. rep. AIIM D110. Written By Gartner for AIIM International, Apr. 30, 2001 (cit. on p. 102).

[154] *Event Sourcing Basics.* 2015. URL: http://docs.geteventstore.com/introduction/event-sourcing-basics/ (visited on 06/13/2015) (cit. on pp. 108, 109).

[155] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997 (cit. on pp. 109, 110).

[156] Edsger W. Dijkstra. "On the role of scientific thought". Aug. 1974 (cit. on p. 110).

[157] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997 (cit. on p. 110).

[158] E. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004 (cit. on pp. 111, 112, 123).

[159] V. Vernon. *Implementing Domain-Driven Design*. Pearson Education, 2013 (cit. on p. 112).

[160] Greg Young. *CQRS, Task Based UIs, Event Sourcing*. Feb. 16, 2010. URL: `http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/` (visited on 06/18/2015) (cit. on p. 112).

[161] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999 (cit. on p. 114).

[162] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999 (cit. on p. 114).

[163] *CORBA® Success Stories*. 2011. URL: `http://www.corba.org/success.htm` (visited on 06/18/2015) (cit. on p. 115).

[164] Michael Stonebraker and Joey Hellerstein. “What goes around comes around”. In: *Readings in Database Systems* 4 (2005) (cit. on p. 116).

[165] *The C10K problem*. Archived at: http://www.webcitation.org/6ICibHuyd. July 21, 2011. URL: `http://www.kegel.com/c10k.html` (visited on 06/18/2015) (cit. on p. 117).

[166] Douglas C. Schmidt. “Evaluating Architectures for Multithreaded Object Request Brokers”. In: *Commun. ACM* 41.10 (Oct. 1998), pp. 54–60. DOI: `10.1145/286238.286248` (cit. on p. 117).

[167] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. “The Design of the TAO Real-time Object Request Broker”. In: *Comput. Commun.* 21.4 (Apr. 1998), pp. 294–324. DOI: `10.1016/S0140-3664(97)00165-5` (cit. on p. 117).

[168] J.O. Coplien and D.C. Schmidt. *Pattern languages of program design*. Pattern Languages of Program Design t. 1. Addison-Wesley, 1995 (cit. on p. 117).

[169] Edited Jim Coplien, Douglas C. Schmidt, and Douglas C. Schmidt. *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. 1995 (cit. on p. 117).

[170] Douglas C. Schmidt. “ASX: An Object-oriented Framework for Developing Distributed Applications”. In: *Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference - Volume 6*. CTEC’94. Cambridge, MA: USENIX Association, 1994, pp. 12–12 (cit. on p. 118).

[171] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. “Design and Performance of an Object-oriented Framework for High-speed Electronic Medical Imaging”. In: *Proceedings of the 2Nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*. COOTS’96. Toronto, Ontario, Canada: USENIX Association, 1996, pp. 15–15 (cit. on p. 118).

[172] R. Greg Lavender and Douglas C. Schmidt. “Pattern Languages of Program Design 2”. In: ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Active Object: An Object Behavioral Pattern for Concurrent Programming, pp. 483–499 (cit. on p. 118).

[173] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas D. Jordan. *Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*. Tech. rep. Washington University, 1997 (cit. on p. 118).

[174] Mohammed Saeed, Mauricio Villarroel, Andrew T. Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H. Kyaw, Benjamin Moody, and Roger G. Mark. "Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): A public-access intensive care unit database". In: *Critical Care Medicine* 39 (May 2011), pp. 952–960 (cit. on p. 144).

[175] *Apache Accumulo*. URL: `https://accumulo.apache.org/` (visited on 07/28/2015) (cit. on pp. 144, 145).

[176] Paul G. Brown. "Overview of sciDB: Large Scale Array Storage, Processing and Analysis". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 963–968. DOI: `10.1145/1807167.1807271` (cit. on p. 144).

[177] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley Zdonik. "S-Store: A Streaming NewSQL System for Big Velocity Applications". In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1633–1636. DOI: `10.14778/2733004.2733048` (cit. on p. 145).

[178] Michael Stonebraker and Ugur Cetintemel. ""One Size Fits All": An Idea Whose Time Has Come and Gone". In: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 2–11. DOI: `10.1109/ICDE.2005.1` (cit. on p. 145).

[179] Aaron J. Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, Sam Madden, Dave Maier, Timothy Mattson, Stavros Papadopoulos, Jeff Parkhurst, Nesime Tatbul, Manasi Vartak, and Stan Zdonik. "A Demonstration of the BigDAWG Polystore System". In: *Proceedings of the VLDB Endowment* 8.12 (2015) (cit. on p. 145).

[180] *Apache Lucene*. URL: `https://lucene.apache.org/core/` (visited on 07/28/2015) (cit. on p. 145).

[181] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu, and Calisto Zuzarte. "Business-Intelligence Queries with Order Dependencies in DB2". In: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.* Ed. by Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy. OpenProceedings.org, 2014, pp. 750–761. DOI: `10.5441/002/edbt.2014.81` (cit. on p. 150).

[182] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Przemyslaw Pawluk, and Calisto Zuzarte. "Queries on Dates: Fast Yet Not Blind". In: *Proceedings of the 14th International Conference on Extending Database Technology*. EDBT/ICDT '11. Uppsala, Sweden: ACM, 2011, pp. 497–502. DOI: `10.1145/1951365.1951424` (cit. on pp. 150, 152).

[183] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. "Fundamentals of Order Dependencies". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1220–1231. DOI: `10.14778/2350229.2350241` (cit. on p. 150).

[184] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. "Expressiveness and Complexity of Order Dependencies". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1858–1869. DOI: `10.14778/2556549.2556568` (cit. on p. 150).

[185] *Spring Framework*. URL: `http://projects.spring.io/spring-framework/` (visited on 07/28/2015) (cit. on p. 154).

[186] Michał Chromiak and Zdzislaw Lojewski. "Stream security particularities in Java". In: *Annales UMCS, Informatica* 8.1 (2008), pp. 5–13. DOI: `10.2478/v10065-008-0001-9` (cit. on p. 155).

[187] *ISO/IEC 9075-1:1999*. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26196 (visited on 07/28/2015) (cit. on p. 157).

[188] *ISO/IEC 9075-2:1999*. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26197 (visited on 07/28/2015) (cit. on p. 157).

[189] Michał Chromiak and Krzysztof Stencel. "The Linkup Data Structure for Heterogeneous Data Integration Platform". In: *Proceedings of the 4th International Conference on Future Generation Information Technology*. FGIT'12. Gangneug, Korea: Springer-Verlag, 2012, pp. 263–274. DOI: 10.1007/978-3-642-35585-1_36 (cit. on pp. 157, 161).

[190] Michał Chromiak and Krzysztof Stencel. "A Data Model for Heterogeneous Data Integration Architecture". In: *Beyond Databases, Architectures, and Structures - 10th International Conference, BDAS 2014, Ustron, Poland, May 27-30, 2014. Proceedings*. Ed. by Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek, and Daniel Kostrzewa. Vol. 424. Communications in Computer and Information Science. Springer, 2014, pp. 547–556. DOI: 10.1007/978-3-319-06932-6_53 (cit. on p. 157).

[191] Piotr Przymus, Aleksandra Boniewicz, Marta Burzanska, and Krzysztof Stencel. "Recursive Query Facilities in Relational Databases: A Survey". In: *Database Theory and Application, Bio-Science and Bio-Technology - International Conferences, DTA and BSBT 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*. Ed. by Yanchun Zhang, Alfredo Cuzzocrea, Jianhua Ma, Kyo-Il Chung, Tughrul Arslan, and Xiaofeng Song. Vol. 118. Communications in Computer and Information Science. Springer, 2010, pp. 89–99. DOI: 10.1007/978-3-642-17622-7_10 (cit. on p. 158).

[192] Marta Burzańska, Krzysztof Stencel, Patrycja Suchomska, Aneta Szumowska, and Piotr Wiśniewski. "Recursive Queries Using Object Relational Mapping". In: *FGIT*. Ed. by Tai-Hoon Kim, Young-Hoon Lee, Byeong Ho Kang, and Dominik Ślęzak. Vol. 6485. Lecture Notes in Computer Science. Springer, 2010, pp. 42–50 (cit. on p. 158).

[193] Aneta Szumowska, Marta Burzańska, Piotr Wiśniewski, and Krzysztof Stencel. "Efficient Implementation of Recursive Queries in Major Object Relational Mapping Systems". In: *FGIT*. 2011, pp. 78–89 (cit. on p. 158).

[194] Aneta Szumowska, Marta Burzańska, Piotr Wiśniewski, and Krzysztof Stencel. "Extending HQL with Plain Recursive Facilities". In: *ADBIS (2)*. Ed. by Tadeusz Morzy, Theo Härder, and Robert Wrembel. Vol. 186. Advances in Intelligent Systems and Computing. Springer, 2012, pp. 265–272 (cit. on p. 158).

[195] Piotr Wiśniewski, Aneta Szumowska, Marta Burzańska, and Aleksandra Boniewicz. "Hibernate the Recursive Queries - Defining the Recursive Queries using Hibernate ORM". In: *ADBIS (2)*. Ed. by Johann Eder, Mária Bieliková, and A Min Tjoa. Vol. 789. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 190–199 (cit. on p. 158).

[196] Ahmad Ghazal, Alain Crolotte, and Dawit Yimam Seid. "Recursive SQL Query Optimization with k-Iteration Lookahead". In: *DEXA*. Ed. by Stéphane Bressan, Josef Küng, and Roland Wagner. Vol. 4080. Lecture Notes in Computer Science. Springer, 2006, pp. 348–357 (cit. on p. 159).

[197] Marta Burzańska, Krzysztof Stencel, and Piotr Wiśniewski. "Pushing Predicates into Recursive SQL Common Table Expressions". In: *ADBIS*. Ed. by Janis Grundspenkis, Tadeusz Morzy, and Gottfried Vossen. Vol. 5739. Lecture Notes in Computer Science. Springer, 2009, pp. 194–205 (cit. on p. 159).

[198] Carlos Ordonez. "Optimization of Linear Recursive Queries in SQL". In: *IEEE Trans. Knowl. Data Eng.* 22.2 (2010), pp. 264–277 (cit. on p. 159).

[199] *Neo4J – Graph Database*. URL: `http://neo4j.com/` (visited on 07/28/2015) (cit. on p. 159).

[200] *Spring Framework JDBCTemplate Documentation*. URL: `http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html` (visited on 07/28/2015) (cit. on p. 159).

[201] Michael Hunger. *Load CSV with success*. Accessed: 2015-02-06. 2014 (cit. on p. 159).

[202] Neo4j. *LOAD CSV into Neo4j quickly and successfully*. Accessed: 2015-02-06. 2014 (cit. on p. 159).

[203] Mike Bergman. *Big Structure and Data Interoperability*. Aug. 18, 2014. URL: `http://www.mkbergman.com/1782/big-structure-and-data-interoperability/` (visited on 05/02/2015) (cit. on p. 180).