
**Bulgarian Academy of Sciences**
**Institute of Information and Communication Technologies**

Jens Kohler

# Optimizing Query Strategies in Fixed Vertical Partitioned and Distributed Databases and their Application in Semantic Web Databases

Doctoral Thesis

Doctoral Program: Informatics

Professional Area: 4.6 Informatics and Computer Science

Supervisor: Prof. Dr. Kiril Simov

Sofia, 2017

# Table of Contents

# List of Abbreviations

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durablity |
| API | Application Programming Interface |
| BASE | Basically Available, Soft State, Eventual Consistent |
| BGP | Basic Graph Pattern |
| CDSA | Common Data Security Architecture |
| CIA | Confidentiality, Integrity, Availability |
| CIMI | Cloud Infrastructure Management Interface |
| CRUD | Create, Read, Update, Delete |
| DBaaS | Database as a Service |
| DSR | Design Science Research |
| eIDAS | Electronic Identification and Trust Services Regulation |
| ENISA | European Union Agency for Network and Information Security |
| ERM | Entity Relationship Model |
| EU | European Union |
| FVPD | Fixed Vertical Partitioning and Distribution |
| HDD | Hard Disk Drive |
| HQL | Hibernate Query Language |
| IaaS | Infrastructure as a Service |
| IDE | Integrated Development Environment |
| I/O | Input/Output |
| IRI | Internationalized Resource Identifier |
| JDBC | Java Database Connectivity |
| JPQL | Java Persistence Query Language |
| JSR | Java Specification Request |
| KEGG | Kyoto Encyclopedia of Genes and Genomes |
| LD | Linked Data |
| LFU | Least Frequently Used |
| LOD | Linked Open Data |
| LRU | Least Recently Used |
| NIST | National Institute of Standards and Technology |
| OASIS | Organization for the Advancement of Structured Information Standards |
| OCCI | Open Cloud Computing Interface |
| OBDA | Ontology Based Data Access |
| OECD | The Organisation for Economic Co-operation and Development |

| | |
|---|---|
| OGM | ObjectGrid Mapper (e.g. Hibernate) |
| OID | Object Identifier |
| OLAP | Online Analytical Processing |
| OLTP | Online Transactional Processing |
| ORM | Object Relational Mapping/Object Relational Mapper |
| OO | Object-oriented |
| PaaS | Platform as a Service |
| PC | Personal Computer |
| PKI | Public Key Infrastructure |
| QoS | Quality of Service |
| R2RML | Relational Database to Resource Description Framework Mapping Language |
| RAM | Random Access Memory |
| RDB | Relational Database |
| RDF | Resource Description Framework |
| SaaS | Software as a Service |
| *SeDiCo* | A Framework for a Secure and Distributed Cloud Data Store |
| SLA | Service Level Agreement |
| SPARQL | SPARQL Protocol And RDF Query Language |
| SQL | Structured Query Language |
| SSD | Solid State Drive |
| SSL | Secure Socket Layer |
| TERC | Trust, Eco, Risk, Cost |
| TOSCA | Topology and Orchestration Specification for Cloud Applications |
| TLS | Transport Layer Security |
| TPC | Transaction Processing Performance Council |
| TTL | Time to Live |
| UML | Unified Modeling Language |
| VPN | Virtual Private Network |
| WWW | World Wide Web |
| XML | Extensible Markup Language |
| YCSB | Yahoo! Cloud Serving Benchmark |

# Introduction

Storing data in relational databases has a long history since Codd defined the relational model and its normal forms in (Codd, 1970). Such relational databases still build the foundation for various applications throughout all application domains even with todays growing data volumes. It is assumed that, despite a rapid dissemination of In-Memory or NoSQL databases, relational databases will keep their important role.

Hence, also relational databases are used as a foundation to store huge volumes of data and this is exactly where Cloud Computing offers dynamic and scalable capabilities. Renting such technological assets and capabilities from external cloud providers is an interesting approach. The pay-as-you-go character of these cloud offers, promise the usage of computing assets without large initial investments. In Cloud Computing environments, dedicated services are used for a certain time and are paid only for the respective usage. Moreover, as there are dedicated services, the complexity to integrate and use them is considered lower compared to paradigms like service-oriented architectures.

As there are still open **data security** and **data protection** challenges, the usage of especially public Cloud Computing is far behind the expectations of e.g. Gartner (Carlton, 2013) and IDC (Gens & Shirer, 2013). Thus in this thesis, **data security** and **data protection** challenges for **relational databases** are addressed with the definition and an implementation of a framework for a ***SE*cure and *DI*stributed *C*loud Data St*O*re**, exploiting a *fixed vertical partitioning and distribution* (FVPD) scheme. ***The main contribution of this work is to show that the proposed framework provides comparable response times to non-partitioned relational databases using cloud infrastructures and contemporary hardware devices.***

# Importance of the Topic

An approach that contributes to the broad dissemination of using especially public clouds is *SeDiCo*, a framework for a *SE*cure and *DI*stributed *C*loud Data St*O*re. The key concept of this approach is to vertically partition relational database data and store the respective partitions in different databases operated in different clouds. *SeDiCo* enables users to define vertical partitions individually, but once a partition scheme is developed, it is not possible to modify it without defining an entirely new one. Thus, the partitioning is called *fixed vertical partitioning and distribution* (FVPD) scheme in the remainder of this work. The author of this work firstly proposed this so-called *Security-by-Distribution* concept in 2012 (Kohler & Specht, 2012) and developed and implemented it prototypically[1] from 2012 to 2014 (Kohler & Specht, 2014a). Although these works proved the technological feasibility, the approach still suffers from severe performance problems when the partitioned and distributed data are accessed. These performance issues are in the focus of this thesis, which aims at investigating, developing and evaluating new ways of accessing those data. In order to not exceed the limits of this work, this thesis focuses on the response time. On the one hand, recent analyses of the author show that the insert, update, and delete operations are also affected (Kohler & Specht, 2014b) (Kohler & Specht, 2014c). On the other hand, (Krueger et al., 2010) showed that $\sim 90\%$ of all operations in enterprise databases are *queries* (i.e. *selects*). Hence, the focus of this work is on the response time of a query and the insert, update and delete performance are considered as key questions of future work tasks. Above that, it can be stated that the usage of Cloud Computing capabilities still are a weighing between security and performance and this thesis aims at minimizing this gap with the definition and the evaluation of adequate query strategies.

A motivating example of the entire *SeDiCo* framework is drawn in Figure 1 which illustrates the FVPD (*fixed vertical partitioning and distribution*) approach (in the remaining part of the thesis this approach is also referred to as *Security-by-Distribution* approach) with a simple *CUSTOMER* relation.

In this example, there are 2 vertical partitions one containing more sensitive data (*Customer_Partition1*) and less sensitive data (*Customer_Partition2*). The

[1] In close cooperation with the theses supervised by the author listed at the end of the thesis. Moreover, *SeDiCo*, as well as the query mechanisms developed in this thesis are generally available under GPL-License at: http://github.com/jenskohler

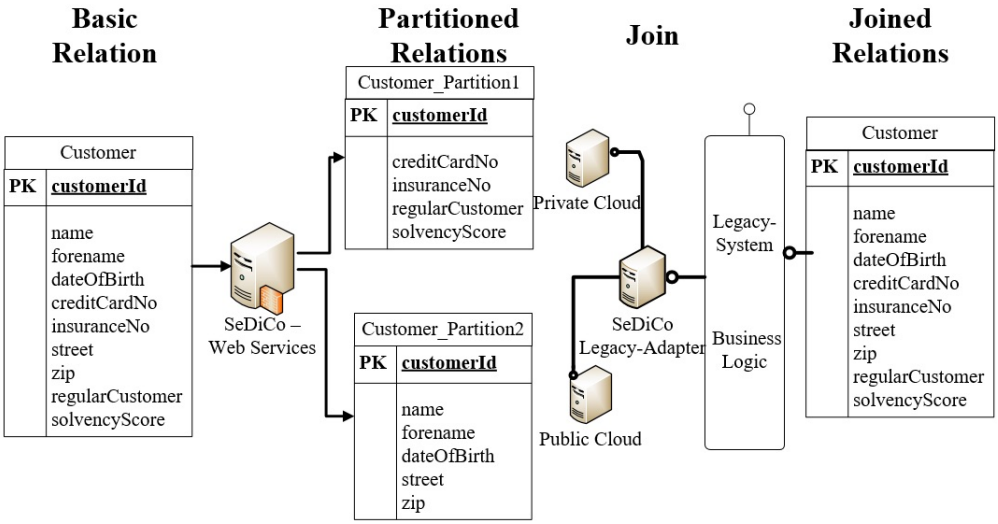



Figure 1: Motivating *SeDiCo* Example

basic idea is now that an intruder is not able to reconstruct entire *CUSTOMER* rows, since the partitioning and distribution scheme is unknown to him. Therefore, it is of minor importance which data are stored in which cloud (public, private, community, hybrid) respectively.

# Overview of the Main Results in the Area

The presented version of *SeDiCo* (cf. Chapter 2) was developed and implemented before the work on the thesis has started. The results of this preliminary work have shown that the ideas behind *SeDiCo* are feasible and work in practice. However, there is still an open question regarding the framework performance in practical use cases:

- **Performance optimization of the *SeDiCo* approach:** Although the feasibility of the original implementation is empirically shown and formally proved, the response time (especially for larger data sets, i.e. more than 10K rows) decreased tremendously. Thus, how can the response time for a FVPD query in practical use cases scenarios be improved, such that it is in the same order of magnitude as a non-FVPD query?

*To the best of the author's knowledge, no one has followed a vertical database partitioning approach in the context of data security and privacy yet. Hence, this thesis conceptualizes, implements and evaluates advanced query mechanisms in order to improve the overall response time of* SeDiCo.

Current figures of the initial implementation can be found in the author's previously published work, e.g. (Kohler, Simov, & Specht, 2015) (Kohler & Specht, 2014b) and in Chapter 6.3.

*All in all, this thesis uses these figures as a basic performance metric and compares the investigated advanced query mechanisms to it.*

Previous works (e.g. (Son & Kim, 2004) (Grund et al., 2011) (Li & Gruenwald, 2012) (Rodríguez & Li, 2011)) on vertical database partitioning have been conducted in the context of performance optimization tasks. Here, databases are (dynamically) vertically partitioned, such that queries do not have to iterate through entire data sets, but only on e.g. those data, that are used very often. Hence, these optimization approaches are workload driven, i.e. the approach depends on the queries issued against the database[2]. Regarding these approaches, *SeDiCo* is different as it follows a *fixed* vertical partitioning approach, in which data are partitioned not according to database workloads, but according to security and privacy related issues. Therefore, in *SeDiCo* it holds that once a partitioning scheme is defined, it is not changeable (during runtime) anymore.

Another interesting field of research with respect to the vertical partitioning and distribution approach is *Cloud Computing*. The *cloud* offers dynamically scalable computing capabilities (i.e. CPU, storage, etc.). It also offers the possibility to rent capabilities from 3rd party cloud providers, such that no infrastructure investments are necessary anymore. However, renting resources from an (possibly) unknown cloud provider requires a great demand of *trust*. Closely aligned to *trust* are *data security* and *privacy* issues. These issues are the main reason why the usage of the cloud is far behind its expectations (e.g. Gartner (Carlton, 2013) and IDC (Gens & Shirer, 2013)).

*With respect to this, the thesis with its FVPD approach proposes the possibility to partition and distribute data, such that each of a certain amount of different cloud providers only gets a logically independent data chunk, which is not usable without the others. Thus, the FVPD approach fosters the usage of (possibly untrustworthy public) Cloud Computing, which is a promising alternative to huge investments in IT infrastructures. Furthermore, based on the FVPD approach, the thesis maps the FVPD approach to the well-known CIA-Principles (confidentiality, integrity, and availability), defined by (DIN ISO 27000, 2011), and illustrates how the level of security and privacy is improved.*

[2]which is the main reason why these works are not considered in more detail in this thesis

# Goals and Tasks of the Thesis

The response time evaluation of the initial *SeDiCo* implementation (Kohler & Specht, 2014b) and (Kohler & Specht, 2014c) showed that there is a tremendous performance loss (factor ∼460 considering the average response time) with the vertical partitioning and distribution approach. However, with an advanced level of data security and privacy (Kohler & Specht, 2015a), this approach enables the usage of public cloud infrastructures. This shows that the *SeDiCo* approach is still a weighing between security and performance.

***Hence, the objective of this thesis is to find strategies, concepts and corresponding implementations to improve the response time to a level that it is in the same order of magnitude as a non-partitioned and non-distributed approach. This results in a minimization problem of the required time to retrieve the result set of a certain query (i.e. the response time) that is issued against* fixed vertically partitioned and distributed *(FVPD) data.***

With respect to this, the hypotheses that are investigated can be formulated as follows:

**Hypothesis 0:** The definition of a Fixed Vertically Partitioned Schema (FVPD) for relational databases improves the level of data security and data protection by separating (i.e. partitioning) and distributing logically coherent data to different storage locations.

**Hypothesis 1:** Query Rewriting improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to partitioned and parallelized query and join implementations.

**Hypothesis 2:** Caching data improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to the usage of In-Memory caches.

**Hypothesis 3:** Using Solid State Disks (SSDs) as distributed secondary storage devices for the FVPD data improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to faster access times of the memory.

Based on the hypotheses, the following tasks are conducted:

**Task 1:** the definition of a methodology for creating an FVPD schema for relational data and a proof of the correctness of the methodology;

**Task 2:** the conceptualization of adequate query mechanisms for relational FVPD data sets;

**Task 3:** the implementation of these relational query mechanisms in Java;

**Task 4:** the evaluation of these relational query mechanisms in terms of their response time;

**Task 5:** the comparison of all developed relational query mechanisms against each other and against the initial *SeDiCo* implementation;

**Task 6:** the application of the FVPD methodology in the Semantic Web with Resource Description Framework-based (RDF-based) data.

Tasks 1-5 are performed to either prove or to reject the previously formulated hypotheses. More specifically, in order to evaluate the response time of the FVPD-implementing query mechanisms, the response time of a non-partitioned and non-distributed data set is used as a basic foundation. *The overall goal is to find approaches (i.e. query mechanisms) whose response times are in the same order of magnitude as the response times based on a non-partitioned and non-distributed data set.* Therefore, these approaches are implemented and evaluated with the TPC-W benchmark (TPC, 2003) in order to remain comparable with previous works in the context of *SeDiCo* (Kohler & Specht, 2015c) (Kohler, Simov, Fiech, & Specht, 2015) (Kohler & Specht, 2015a) (Kohler, Simov, & Specht, 2015). Moreover, an overview about all developed query mechanisms and their response time is presented and compared to a traditional non-FVPD approach.

Task 6 is conducted to show the generic character of the entire *SeDiCo* framework and that the basic concepts and approaches are (partly) also viable in other application domains. With respect to RDF-based data, it must be noted that the approach illustrated in this thesis is restricted to the so-called closed world assumption, in which a *complete* data set is exposed as RDF-based data and this RDF-based data can be queried via SPARQL (SPARQL Protocol And RDF Query Language) language. Again, the approach is restricted to SPARQL queries (analogous to the relational approach), that refer to *complete* data sets and to queries that always terminate.

Finally, the expected results can be subsumed as follows:

**Result 1:** a formal correctness proof of the FVPD methodology;

**Result 2:** ready-to-use FVPD query execution methods;

**Result 3:** an evaluation of the query mechanisms that acts as a guideline for their concrete application in different scenarios;

**Result 4:** a classification of the query mechanisms, which ones are applicable in which scenarios;

**Result 5:** a conceptual transfer of the relational FVPD approach to other application domains (i.e. the Semantic Web with RDF-based data) to emphasize the generic character of the approach;

**Result 6:** a demonstration of how the entire *SeDiCo* approach can be applied in the Semantic Web on RDF-based data.

## Contributions of the Thesis

With the successful implementation and evaluation of the before-mentioned tasks, the thesis contributes to the current state-of-the-art with the following aspects.

**Contribution 1:** Definition of a *Security-by-Distribution* Principle for Relational Databases

In this thesis, there is a *Security-by-Distribution* principle introduced that uses vertical relational database partitioning to logically separate database tables into chunks that are worthless without the others, but can be joined based on the containing primary key. This principle is used in the so-called *SeDiCo* framework. The respective chunks are distributed (ideally) across different clouds and only the user who partitioned and distributed the rows knows the partitioning distribution scheme of the partitions (chunks). This increases the level of security and privacy and enables the storage of data in especially public cloud infrastructures.

**Contribution 2:** Development of FVPD Query Strategies

The previously mentioned *Security-by-Distribution* approach requires new ways of accessing the partitioned and distributed rows, as they have to be joined, i.e. entirely reconstructed before they are actually accessible. All

approaches are conceptualized, implemented and evaluated in the presented thesis.

**Contribution 3:** FVPD Query Strategy Integration into the *SeDiCo* Framework

This thesis is created in the context of the *SeDiCo* framework development. As a further result, the approaches conceptualized and illustrated in this thesis are implemented and positively evaluated ones are integrated into the framework. This will develop the entire framework to a feasible opportunity in practical usage scenarios, which will allow further performance analyses in various application domains, where relational databases build the foundation for applications.

**Contribution 4:** FVPD Performance Evaluation and Classification

The developed query mechanisms are evaluated with respect to their response time and compared to each other to provide a short but precise overview about all investigated approaches and their respective response time.

**Contribution 5:** Transfer of the FVPD Methodology to other Databases

Here, the entire *SeDiCo* approach is transferred to a Semantic Web scenario, based on the *resource description framework* (RDF). Firstly, this demonstrates the universal application character of the basic approach[3] and secondly, it proves that the approach can be transferred and applied to other application domains with a clearly stated and demonstrated integration effort.

# Methodology Used for the Research

The overall research focus aims at developing and improving a framework to demonstrate that using especially public clouds is not per se insecure. Therefore, the goal of this thesis is to develop *query strategies* (i.e. so-called *artifacts* according to (Hevner et al., 2004)) that considerably increase the performance of vertically partitioned and distributed databases operated in several clouds, such

[3]other thinkable application scenarios could involve NoSQL datastores with its four fundamental architectures (column, document, key-value stores and graph databases)

that the entire FVPD approach becomes comparable to a non-partitioned and non-distributed one.

An *artifact* can be defined analogously to (Hevner et al., 2004) as a piece of software (*[...] something that is artificial, or constructed by humans, as opposed to something that occurs naturally.).* Furthermore, *artifacts* must *[...] improve existing solutions to a problem or perhaps provide a first solution to an important problem* (Hevner & Chatterjee, 2010).

In order to answer the research question, the *Design Science Research* (DSR) methodology described by Hevner et al. is used (Hevner & Chatterjee, 2010). The aim is to *extend boundaries of human and organizatorial capabilities by creating new and innovative artifacts* (Hevner et al., 2004). This methodology provides a 7-step framework to transfer research work into concrete (enterprise) applications and is illustrated in Figure 2[4].

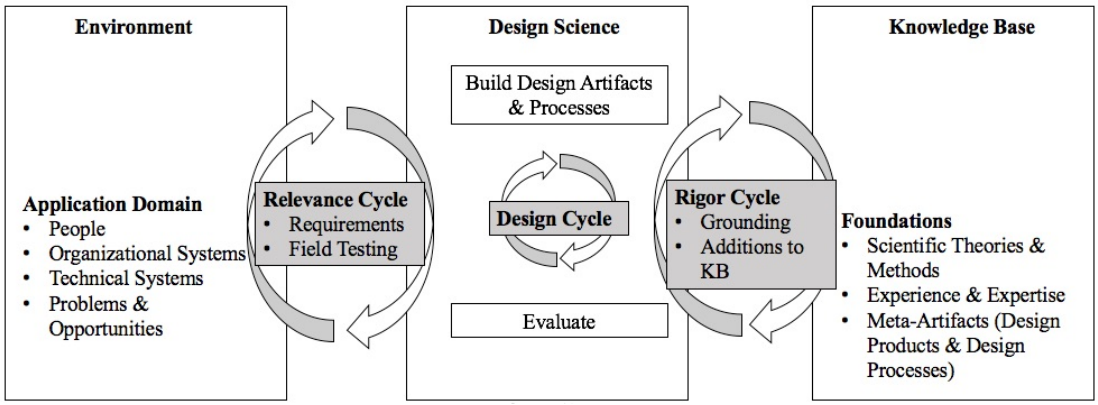



Figure 2: Design Science Research Cycles

This strategy was chosen because it perfectly targets the framework development process of *SeDiCo* and its applied science character. Based on the results of this work, further research projects in cooperation with partners from industries and other research groups in national as well as international contexts should be acquired.

The entire *SeDiCo* framework development and its associated research work are aligned to these DSR Cycles. To illustrate this in more detail, Figure 3 maps the DSR Cycles to the presented thesis.

[4]adapted from (Hevner & Chatterjee, 2010)

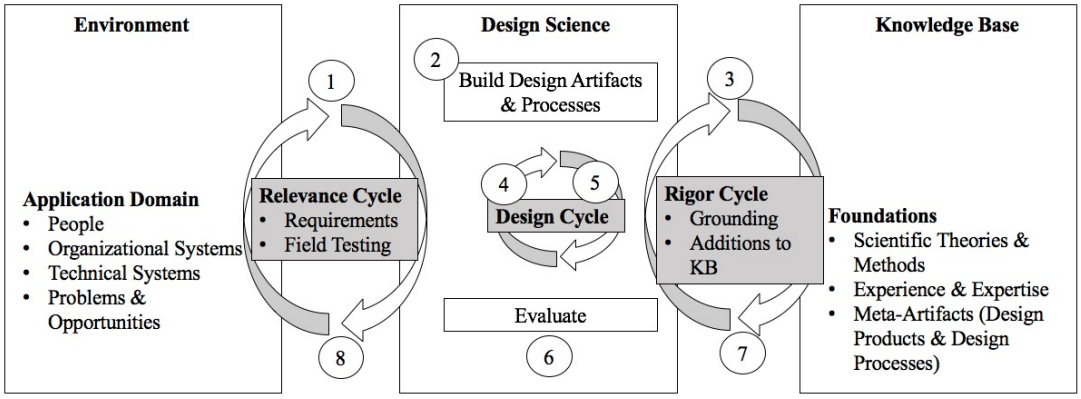



Figure 3: Design Science Research Cycle Mapped to Thesis Chapters

This thesis outlines *SeDiCo*, a framework that implements the FVPD methodology and therefore in the rest of the thesis does not use the term *artifact*, but uses *query strategy*, *method* or *approach* in order to stick with the term *FVPD methodology*.

# Scope and Limitations

The actual *SeDiCo* framework is developed in a broader sense with funded research projects (mentioned in the respective author's publications at the end of the thesis), and student works, supervised by the author (also mentioned at the end of the thesis). These other works deal with mechanisms and concepts that abstract from different cloud or database implementations and encapsulate them in uniquely accessible interfaces. Other works concentrate on concrete implementation tasks (e.g. the FVPD partitioning or the join of the FVPD data). In order to narrow the focus of this thesis down to a concrete specific research question, it focuses on the improvement of the framework's response time.

Because of the before-mentioned predominance of analytical database workloads (i.e. select queries) the evaluation is restricted to the response time. This work aims at finding *query strategies* that are suitable for FVPD data and finding concepts and approaches that are worth pursuing any further. Therefore, performance evaluations that take the response time into consideration are conducted and the response time of queries is considered as an aspect of greater importance compared to insert, update and delete operations, especially as ∼90% of all operations in enterprise databases are queries.

Above that, this work aims at providing a basic response time measurement which acts as a foundation for further investigations. Hence, the theoretical lower and upper bound for the response time of the entire FVPD approach is experimentally proved. To achieve this, the evaluation uses a *projection* query[5]. Yet, it can be noted that *SeDiCo* and all developed query strategies in this work support the full SQL standard (ISO/IEC, 2011) including further research work with respect to additional response time optimization opportunities.

A detailed security and privacy evaluation of the entire *SeDiCo* framework would be too specific for each application domain and therefore, this task is postponed to a stage in which the framework is disseminated to a broader public.

# Formal Conventions

This section briefly defines the formal conventions for citations, notions, emphases and the used source code, to facilitate the reading and to understand the usage of these conventions.

## Citations

Direct and indirect citations are written in italic letters directly followed by the respective reference. Furthermore, the *Harvard Style Notation* is used throughout this thesis for the references.

Short references within the text are a combination of the authors or authors names followed by the year of the publication surrounded by parentheses, e.g. (Codd, 1970).

The complete reference list can be found at the end of this thesis.

[5] i.e. SELECT * FROM *tablename*; As the partitioning and distribution according to the FVPD approach splits the table based on its attributes, the primary key has to be replicated in all partitions, such that it can be used afterwards for the join of the partitions. For the sake of better readability, the presented thesis illustrates the FVPD approach and its corresponding *join* with 1 primary key. It further has to be noted that the approach works analogously with $n$ primary keys (i.e. *compound primary keys*). In such cases, all primary key attributes must be replicated in all partitions.

## Notations

*Data*, as stated in Oxford Dictionaries is the plural form of datum. Accordingly, it is also used as a plural noun in English in this thesis (although nowadays also accepted as singular), unless it is followed by another singular noun, e.g. data set or data volume.

The plural for *indexes* is written as *indices* according to the Oxford Dictionary recommendation.

Notions that are abbreviated are written in full when they are introduced. Then the abbreviation is written directly after the newly introduced notion in brackets, e.g. Secure and Distributed Cloud Data Store (*SeDiCo*). The complete list of abbreviations can be found at the beginning of this thesis.

## Text Emphases

Text emphases of key concepts or notions of particular importance are written in italic letters to underline their importance, e.g. *SeDiCo*. Unlike citations these notions are neither included in quotation marks nor followed by a reference as they are formally defined before their actual usage.

## Formulas

Mathematical formulas are written as the following example shows:

$$RS \leftarrow \Pi_{(a_1,\ldots a_n)} R(A)$$

## Source and Pseudo Code

The source code used in this thesis is presented as Java code and the used pseudo code is a slightly adapted version of Java code. Both kinds of code are presented as so-called code listings with line numbers at the beginning of each new line. Thus, lines of particular importance can be referenced in the text more easily, e.g.

Listing 1: Example Source/Pseudo Code

```
1 public static void main(String[] args) {
2         System.out.println("Hello World");
3 }
```

# Structure of the Thesis

This section outlines the entire thesis structure and summarizes its main goals, the research problem and the expected results. Above that, formal conventions and the contribution to the state-of-the-art are presented. Chapter 1 covers the definition of the research problem and notions that are used throughout the work. Basic definitions are given, the hypotheses are formulated and the expected results of the investigated query mechanisms are outlined. After that, Chapter 2 defines the FVPD methodology formally and proves its correctness, afterwards the original *SeDiCo* framework implementation without any optimized query strategies is illustrated. Chapter 3 relates the key topics of this thesis to current state-of-the-art research works. In Chapter 4, the query strategies are conceptualized and designed and their implementation is presented in Chapter 5. After all, the query strategies are evaluated in Chapter 6 which is followed by a conclusion in Chapter 7. Then, Chapter 8 gives an outlook about future application scenarios of the FVPD approach, where a Resource Description Framework (RDF)-based database is vertically partitioned and distributed. Finally, the last chapter summarizes the entire thesis and provides an outlook about interesting and relevant work concerning the future *SeDiCo* framework development.

The target group for this thesis is primarily researchers and practitioners with a strong relation to database systems and architectures, who aim at using (theoretically unlimited scalable) cloud capabilities for the storage or analysis of data. It further addresses researchers and practitioners in the field of *Cloud Computing*, who aim at finding an approach to store data securely and privacy-aware in a distributed cloud environment with different deployment and service models involved.

# Chapter 1

# Problem Definition

This chapter starts with formal definitions of central notions and general concepts and their adaptions to the context of the presented work. Having outlined the basic definitions, the research problem and the derived hypotheses are presented and formalized. After that, the expected results of this work are presented.

## 1.1 Data Set Definition

This section outlines the formal problem definition of the research problem addressed in this thesis. Here the main notions used within the thesis are introduced. The starting point is a *database table* commonly known as a *relation* from Codds relational model (Codd, 1970). A *relation* $R$ with a set of attributes $A = \{a_1, a_2, \ldots, a_n\}$ is denoted as $R(A)$. The ordering of the attributes in the definition of the relation is important in the context of the thesis. Therefore, the set of attributes for a relation is an ordered list with a possible repetition of some attributes. The set of attributes refers to the *table headers* for the relation[1]. The *number of attributes* $n$ represents the *degree of a relation*. The relation of degree $n$ is called $n - ary$ relation. The table representation of a relation is a set of *rows*, where each column corresponds to an attribute in the table header. Each *row* represents a *tuple* of values for the corresponding attributes called *attribute values* (in the thesis called *values*). The number of the rows in a relation is called *cardinality* of the relation, denoted as $|R(A)|$. A relational database consists of one

[1]If there is a set of attributes $A$ represented in two different table headers $A'$ and $A''$, the two relations $R(A')$ and $R(A'')$ are different, although it could be the case that they represent the same information.

or more relations which can share attributes. The concepts of the thesis consider only one relation for the sake of better readability. These concepts are illustrated in Figure 1.1.

In Figure 1.1 the first *row* is the *header of the table* containing the attribute names. The degree of the table is $n$. The *cardinality* of the relation is $j$. Each cell $r_{kl}$ has an *attribute value* for the attribute $k$ in row $l$ with $1 \leq k \leq n$ and $1 \leq l \leq j$.

In order to uniquely identify a certain row $r_l$, there is the concept of a *primary key*. A *primary key* $A_k$ is a set of one or more attributes ($A_k \subseteq A$), such that the attribute values for the attributes in $A_k$ are unique for every row $r$ in $R(A)$. For the sake of better readability, this thesis focuses on relations with a primary key containing just one attribute[2].

R(A)

| attributes $a_1 ... a_n$ | $a_1$ | $a_2$ | $a_3$ | ... | $a_n$ |
|---|---|---|---|---|---|
| row $r_1$ | $r_{a1,1}$ | $r_{a2,1}$ | $r_{a3,1}$ | ... | $r_{an,1}$ |
| row $r_2$ | $r_{a1,2}$ | $r_{a2,2}$ | $r_{a3,2}$ | ... | $r_{an,2}$ |
| row $r_3$ | $r_{a1,3}$ | $r_{a2,3}$ | $r_{a3,3}$ | ... | $r_{an,3}$ |
| | ... | ... | ... | ... | ... |
| row $r_j$ | $r_{a1,j}$ | $r_{a2,j}$ | $r_{a3,j}$ | ... | $r_{an,j}$ |

Figure 1.1: Relational Model

Retrieval of information from a relation is done via a *query* which is evaluated with respect to the relation. The results from such a query execution is called *result set* $RS = \{r_{aj,l}, ..., r_{ak,l}\}$ (with $1 \leq j, k \leq n$, cf. Figure 1.1). The result set is a set or subset of the *rows* in a *relation*[3] that satisfied the conditions stated in the query.

## 1.2 Data Access

In order to access data in a relational database, different operators over the attributes (i.e. *projection*) and rows (i.e. *selection*) are performed. Data (rows

[2]This is not a loss of generality because the algorithms presented in the thesis can be extended to relations with primary keys that consist of more than one attribute.

[3]For some queries the rows in the result set are shorter than in the original relation, e.g. containing no values for some of the attributes

in a relation) in the context of the thesis are accessed with *queries*[4]. Hence, the basic operators used throughout the thesis are introduced here.

### 1.2.1 Selection

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and $R(A)$ be a relation. A *selection* operator determines which *rows* meet the criteria $\varphi$ and which are therefore collected into a *result set* (depicted as $\leftarrow$). *Rows* that do not meet these criteria are omitted. The following *selection* collects all *rows* in a *result set* $RS$ which meet the selection criteria $\omega$ formulated over the relation attributes $\varphi := (a_i = \omega_i, ..., a_j = \omega_j)$ which is issued against a relation $R(A)$:

$$RS \leftarrow \sigma_{(a_i=\omega_i,\ldots,a_j=\omega_j)} R(A) \tag{1.1}$$

with $a_i$ as the $i$th attribute of relation $R(A)$, and $1 \leq i \leq j \leq n$. An example of a *selection*, based on the *CUSTOMER* relation from Figure 1 could be stated as follows:

$$RS \leftarrow \sigma_{(\text{name = 'x' and zip = '123'})} Customer \tag{1.2}$$

The respective SQL implementation would be as follows:

```
SELECT * FROM CUSTOMER WHERE name ='x' and zip = '123';    (1.3)
```

Here, all *attribute values* from tuples in the *CUSTOMER* relation are in the *result set* $RS$, but only those tuples that have an attribute value 'x' for the 'name' *attribute* and that have a value '123' for the 'zip' *attribute*. Hence, a *selection* results in a *horizontal* subset of a *relation* that includes all rows that meet the selection criteria (Elmasri & Navathe, 2015).

[4]either with *selections* or *projections*. The thesis illustrate the basic concepts of the FVPD approach with *projection queries* for the sake of better readability, but it works analogously for *selections* (cf. Sec. 1.2.2). In the FVPD approach, the two operators - *selection* and *projection* - are also similar from a performance point of view with respect to the response time, as the dominating factor for the operators is the *reconstruction* of the original relation.

Moreover, it has to be noted that this work aims at providing a comparable basic query response time measurement for FVPD data. For the sake of better readability[5], the selection criteria $\omega$ are omitted in the following chapters, or explicitly described where necessary.

### 1.2.2 Projection

The next operator relevant for the thesis is a *projection* $\Pi$ over a relation $R(A)$. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and $R(A)$ be a relation. A *projection* is essential for accessing *rows* in a *relation*, as it specifies which *attributes* of the *relation* are collected in the *result set*. Thus, it can be noted that in contrast to the above-mentioned *selection*, a *projection* results in a *vertical* subset of a *relation* (Elmasri & Navathe, 2015).

The following *projection* $\Pi$ collects all *rows* in a *result set* $RS$ that meet the attribute list $(a_i, ...a_j)$ which is issued against a relation $R(A)$:

$$RS \leftarrow \Pi_{(a_i,\ldots,a_j)} R(A) \tag{1.4}$$

with $a_i$ as the $i$th attribute of relation $R(A)$, with $1 \leq i \leq j \leq n$. Thus, since not all attributes are included in this *projection*, only attributes $a_i, ..., a_j$ are collected in the *result set* $RS$.

A corresponding example of this *projection*, based on the *CUSTOMER* relation from Figure 1 could be stated as follows:

$$RS \leftarrow \Pi_{(name,zip)} Customer \tag{1.5}$$

Hence, only the *name* and *zip*[6] of the *Customers* build the *result set* $RS$.

The respective SQL implementation would be:

```
SELECT name, zip FROM Customer;
```
(1.6)

[5] the *SeDiCo* framework supports the full SQL standard
[6] zip codes

It has further to be noted that both query operators (*selection* and *projection*) can be combined to restrict the *result set* accordingly. However, the thesis and especially the benchmark chapter use *projection* queries without any further restrictions[7]. Indeed, a challenging task is the generality of the queries with the various possibilities of combining the query attributes e.g. via *AND*, *OR*, *NOT*, etc. In order to address this challenge, the FVPD approach outlined in this thesis uses the following two mechanisms:

- Firstly, the *primary key attribute* is always added to the respective queries[8]. This ensures that duplicates in the result set are removed, which corresponds to the original definition of a *projection* by (Codd, 1970).

- Secondly, the FVPD approach rewrites the *original query* to so-called *reconstruction queries*[9] that join the partitioned rows to recreate the original ones. After this reconstruction step, the *original query* is issued against the reconstructed rows to maintain the complex attribute combinations. As the *reconstruction queries* recreate the original relation, executing the original query afterwards, is considered a viable approach[10] to be able to generate an adequate solution for general *projections* and *selections*.

Both operators - *selection* and *projection* - are used with respect to a single relation. In addition to this, a *join* operator focuses on processing two or more relations. It combines the result sets of several queries into a single result set. Basically, a *join* can be performed on $n$ relations, but in concrete database implementations this results in the sequential execution of a *join* on two relations. Therefore, the thesis illustrate the *SeDiCo* approach with a *join* on two relations for the sake of better readability, although a *join* over $n$ relations works analogously.

[7] this illustrates the generic character and facilitates the comparison of the measured response times

[8] from an implementation point of view, this is done automatically by the framework by reading the relations' metadata.

[9] always including the primary key attribute

[10] although is leaves room for further optimizations which are considered as future work challenges

### 1.2.3 Join

The *join* operator $\bowtie$ (with $\Theta$ as the join condition[11]) allows the combination of *relations* in a sense that each row from a relation $R$ is joined with a corresponding row in relation $S$. Hence, a *join* $\bowtie$ is defined as follows:

Consider two *relations* $R$ and $S$, each with a set of attributes $A = \{a_1, ..., a_i\}$ and $B = \{b_1, ..., b_m\}$: $R(A)$ and $S(B)$. The *join* of $R(A)$ and $S(B)$ is then defined in the following way:

$$\begin{aligned} R \bowtie_{a_1,...,a_i \Theta b_1,...,b_m} S := \{ & (r_{a_1,k}, ..., r_{a_i,k}) \oplus (s_{b_1,l}, ..., s_{b_m,l}) \mid \\ & R(r_{a_1,k}, ..., r_{a_i,k}) \wedge S(s_{b_1,l}, ..., s_{b_m,l}) \wedge \\ & (r_{a_1,k}, ..., r_{a_i,k}) \; and \; (s_{b_1,l}, ..., s_{b_m,l}) \; meet \; \Theta \} \end{aligned} \tag{1.7}$$

where $r_{a_1,k}$ is the attribute value for the attribute $a_1$ for some row $k$ of the relation $R$, etc., and $s_{b_1,l}$ is the value of the attribute $b_1$ for some row $l$ of the relation $S$. Accordingly, $R(r_{a_1,k}, ..., r_{a_i,k})$ denotes the fact that the list of values $(r_{a_1,k}, ..., r_{a_i,k})$ is a row in the relation $R$. Similarly for $S(s_{b_1,l}, ..., s_{b_m,l})$.

Here, the $\oplus$ denotes a special case of a concatenation of the rows in $R$ and $S$: based on the equality of the *primary key attributes* $a_1$ and $b_1$ respectively, the rows in the relations $R$ and $S$ are merged together.

Thus, Equation (1.7) defines the *join* of the two relations $R$ and $S$ under the condition $\Theta$. The result set of this join can then be stated as $(r_{a_1,k}, ..., r_{a_i,k}, s_{b_1,l}..., s_{b_m,l})$, such that they meet the condition $\Theta$. A further differentiation between the different join types (i.e. theta, natural, and equi join) is not necessary for the developed concepts of this work, as they just indicate which operators are used for the join condition $\Theta$[12].

As already mentioned above, the FVPD approach requires to have the *primary key* ($a_1$) as the common attribute in all FVPD relations. Hence, the equality ($=$)

[11] e.g. depending on which condition is used for $\Theta$ (possible are: $=, \neq, <, >, \leq, \geq$), the join is further distinguished. If the *equality* ($=$) operator is used, it results in a *equi* join, whereas the other mentioned operators all result in a *non-equi join* (Elmasri & Navathe, 2015).

[12] Further kinds of *joins* (i.e. *inner* or *outer* joins) are not considered any further in this work and this also holds for other concepts defined in Codds relational model (e.g. referential integrity constraints). Here the attention is drawn to the relevant literature, i.e. (Elmasri & Navathe, 2015) (Garcia-Molina et al., 2008). Furthermore, for the sake of clarity this work describes the FVPD approach with only two *relations*. However if more *relations* are involved the entire approach works accordingly and this also holds for the *join* of more than two *relations*. In this case a so-called *multi-way join* is performed which is basically a series of a *joins* on two *relations*.

operator (*natural join*) can be used as the join condition $\Theta$ on the *primary key* ($a_1$) and thus, the FVPD *(natural) join* is defined as a consequence from Equation (1.7) as:

$$\begin{aligned} R \bowtie_{a1=b_1} S := \{ & (r_{a_1,k}, ..., r_{a_i,k}) \oplus (s_{b_1,l}, ..., s_{b_m,l}) \mid \\ & R(r_{a_1,k}, ..., r_{a_i,k}) \wedge S(s_{b_1,l}, ..., s_{b_m,l}) \wedge \\ & (r_{a_1,k}, ..., r_{a_i,k}) \; and \; (s_{b_1,l}, ..., s_{b_m,l}) \; meet \; (r_{a_1,k} = s_{b_1,l}) \} \end{aligned} \tag{1.8}$$

Compared to Equation (1.8) a compact notation for the FVPD (natural) join can be reached if all attributes (that are not important for the join condition) are omitted and its results are given in the following definition:

$$R \bowtie_{a_1} S := \{(R) \oplus (S)\} \tag{1.9}$$

### 1.2.4 Naming Conventions

Table 1.1 contrasts the differences between the *theoretical relational model* and the common notions used in most practical database implementations. This approach is also followed in this thesis, where the theoretical parts (e.g. Chapter 1, 3, and 4) use notations from the relational model and Chapter 5, 6, etc. use notations from the practical database implementation.

Table 1.1: Mapping of Relational Model to Database Implementation

| **Relational Model** | **Database Implementation** |
|---|---|
| Relation | Table |
| Tuple or Row | Row |
| Attribute | Column name |
| Attribute $a_i$ | Column name of the $i$th column |
| Degree of relation | Number of attributes |
| Result set | Result set |
| Primary key | Primary key |
| Cardinality | Number of rows |

## 1.3 Problem Formulation

The key approach of this thesis is to create vertical partitions of a relation $R(A)$ and to distribute them across different clouds. For reasons of clarity, the FVPD approach described in this thesis focuses on two vertical partitions $S_v(B)$ and $T_v(C)$. Based on this, the *response time* of this vertical partitioning approach (FVPD) is evaluated.

Vertical partitioning suffers from severe performance degrades since the *join* to recreate the original relation is very time-consuming. In this *join*, it has to be determined which of the attributes belong to which partition and all rows that share the same primary key attribute have to be collected into the result set. Another challenge is the fact that the result set of $S_v(B)$ and $T_v(C)$ cannot be stored at a central location due to security and privacy concerns and it has to be reconstructed with every new query. Lastly, the relevant rows have to be fetched directly from the hard disk, as with the usage of different and distributed database systems, their optimizers, caches, indices, etc. can only improve the response time for every single database but lack any knowledge about the global query processing. Again, such a central logic, based on such global information that would be able to optimize queries would contradict *SeDiCo*'s *Security-by-Distribution* principle.

Hence, the research problem of this thesis can be summed up with finding adequate query strategies that improve the overall response time to a level that is in the same order of magnitude as a query against a non-partitioned and non-distributed database setup. A formal definition of this is a *minimization problem* of the time $t$ required to generate the joint result set based on FVPD relations $S_v(B)$ and $T_v(C)$. This can be stated as follows:

$$min_t(RS_{v\_query_1}S_v(B) \bowtie_{a_1} RS_{v\_query_2}T_v(C))$$

With respect to this minimization problem, a lower bound[13] is the time $t_{lower}$, required to collect the same result set with a non-partitioned relation $R(A)$:

[13]note that the complexity is (in the best case) $\mathcal{O}(n)$ with $n$ as the number of rows in $R$, e.g. if relation $R$ is stored completely in an In-Memory cache,

$$t_{lower} = RS_{query}(R(A))$$

An upper bound for the *response time* is determined by the FVPD *query* itself:

$$t_{upper} \geq (RS_{v\_query_1} S_v(B) \bowtie_{a_1} RS_{v\_query_2} T_v(C))$$

The lower and the upper bounds are determined by the time complexity of the FVPD approach. The dominant factor here is the *join* of the FVPD relations as defined in Section 1.2.3. Therefore, in a naïve approach, this *join* results in the Cartesian Product of the FVPD relations[14], which yields to an upper bound of $\mathcal{O}(n^2)$ with $n$ as the number of rows in the relations and the exponent indicating the number of relations. An analogous consideration can also be made for the lower bound. Again, with the *join* as the predominant factor for the performance of the entire FVPD approach, more sophisticated *join* algorithms are worth considering. A theoretical lower bound with approaches like the *Hash Join*[15] is $\mathcal{O}(n + m)$ with $n$ as building a hash table of all $n$ rows in relation $S$ and $m$ as hashing the corresponding rows of relation $T$ against the hash table[16]. The lower bound for the *Sorted-Merge Join*[17] can be determined as $\mathcal{O}(n + m)$ with $n$ as sorting all $n$ rows in relation $S$ and merging $m$ rows of relation $T$ against this sorted list[18]. The worst-case complexity of the *Sorted-Merge Join*[19], in which none of the relations are sorted, can be stated as $\mathcal{O}(n + m + (n * (log(n) + m * log(m)))$, as here extra sorting effort for the 2 partitions has to be considered as well[20]. Above all, it is assumed that a query against an FVPD data set cannot be faster than the same query against a non-FVPD data set (i.e. a single database relation). This results in an absolute lower bound of $\mathcal{O}(n)$, which can be stated as a database

[14] except that the primary key $a_1$ is not replicated in the result

[15] introduced in Chapter 4 where this approach is actually used

[16] note that $n$ and $m$ denote the cardinality of relations $S$ and $T$ and therefore it follows that $n = m$

[17] introduced in Chapter 4 where this approach is actually used

[18] note that $n$ and $m$ denote the cardinality of relations $S$ and $T$ and therefore it follows that $n = m$

[19] introduced in Chapter 4 where this approach is actually used

[20] as the relations are sorted based on the primary key attribute, the thesis considers the best-case complexity in order to be more comparable to the *Hash Join*

query against a relation containing $n$ rows and all these $n$ rows are collected in the result set.

Both, the lower and the upper bounds could also be determined in concrete figures in experimental setups in (Kohler & Specht, 2014b) for different numbers of rows, ranging from 0 to 288K rows per relation. In these experimental setups the *CUSTOMER* relation from the TPC-W benchmark (TPC, 2003) was used. TPC-W is a database benchmark that emulates various transactions that emerge in a typical web shop, i.e. adding articles to a shopping cart, order articles, etc. TPC is short for *Transaction Processing Performance Council* and is a consortium of various industry partners such as Intel, Mircosoft, Oracle, Red Hat, Cisco, Dell, etc. The basic performance metric of the benchmark evaluates the number of performed transactions per second, which indicates the response time of the web shop. Although the latest benchmark specification is from 2003 and the benchmark is considered deprecated, it is still widely used nowadays and its data model has great similarities with the current TPC-H (TPC, 2014) benchmark from 2014. Since the focus of this work is to present an approach to store data with an improved level of security and privacy in a distributed cloud database architecture, the approach is illustrated with a relation and two corresponding vertical partitions (cf. Figure 1.2). Hence, the *CUSTOMER* relation is regarded as an ideal experimental data model, as it stores personal data (e.g. *credit_card_number*, *date_of_birth*, etc.) for fictional customers.

Above that, this relation has the appealing feature that it can be vertically partitioned and distributed in various ways, such as e.g. in private (credit card, date of birth) and public (*name*, *street*, *zip*) data or shop-centric (*username*, *last_login*) and personal (*name*, *date_of_birth*) data.

Moreover, this relation contains several different data types for its attributes (int, string, time stamp, date, etc.) and a single primary key (*C_ID*) which is considered as an adequate test scenario when different database systems are used. The entire data model can be found in (TPC, 2003) and the CUSTOMER relation with its vertical partitions that are relevant for this work is illustrated in Figure 1.2.

R(A)

| CUSTOMER | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_PASSWD | VARCHAR NOT NULL |
| | C_UNAME | VARCHAR NOT NULL |
| | C_FNAME | VARCHAR NOT NULL |
| | C_LNAME | VARCHAR NOT NULL |
| | C_ADDR_ID | INT NOT NULL |
| | C_PHONE | VARCHAR NOT NULL |
| | C_EMAIL | VARCHAR NOT NULL |
| | C_SINCE | DATE NOT NULL |
| | C_LAST_LOGIN | DATE NOT NULL |
| | C_EXPIRATION | TIMESTAMP NOT NULL |
| | C_DISCOUNT | DOUBLE NOT NULL |
| | C_BALANCE | DOUBLE NOT NULL |
| | C_YTD_PMT | DOUBLE NOT NULL |
| | C_BIRTHDATE | DATE NOT NULL |
| | C_DATA | VARCHAR NOT NULL |

Sv (B)

| CUSTOMER_p1 | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_PASSWD | VARCHAR NOT NULL |
| | C_UNAME | VARCHAR NOT NULL |
| | C_FNAME | VARCHAR NOT NULL |
| | C_LNAME | VARCHAR NOT NULL |
| | C_ADDR_ID | INT NOT NULL |
| | C_PHONE | VARCHAR NOT NULL |
| | C_EMAIL | VARCHAR NOT NULL |
| | C_SINCE | DATE NOT NULL |

Tv (C)

| CUSTOMER_p2 | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_LAST_LOGIN | DATE NOT NULL |
| | C_EXPIRATION | TIMESTAMP NOT NULL |
| | C_DISCOUNT | DOUBLE NOT NULL |
| | C_BALANCE | DOUBLE NOT NULL |
| | C_YTD_PMT | DOUBLE NOT NULL |
| | C_BIRTHDATE | DATE NOT NULL |
| | C_DATA | VARCHAR NOT NULL |

Figure 1.2: Vertical Partitioned TPC-W Customer Relation

## 1.4 Hypotheses

To conclude this section, the following hypotheses are derived from the research problem stated as the minimization of the response time:

$$min_t(RS_{query}S_v(B) \bowtie_{a_1} RS_{query}T_v(C))$$

*Hypothesis 1: Query Rewriting improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to partitioned and parallelized query and join implementations.*

The approach followed to investigate this hypothesis takes the original database query and rewrites it such that it corresponds to the *fixed vertical partitioning* scheme. Performance gains are expected as the vertical partitions are queried individually and only the query-matching rows have to be joined for the final result set. The evaluation of this approach will show if the performance gain is able to reach the response time of the same scenario with a non-partitioned and non-distributed data set. Therefore, this syntactic query rewriting is implemented

and evaluated (in Chapter 4) with 3 different join algorithms, (i.e. *nested-loops*, *hash* and *sorted-merge*). All in all, a database query is issued against a non-FVPD data set and the response time is measured. Then the data set is partitioned with the FVPD approach and the same query is issued against the FVPD data set. Consequently, the query is rewritten and the response time is also measured. Finally, the response times are compared to either prove or reject the hypothesis.

*Hypothesis 2: Caching data improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to the usage of In-Memory caches.*

All in all, there are 3 components that are evaluated with respect to this hypothesis. A basic differentiation is made between the *decentralized* and the *centralized* caching approach. The decentralized approach contains two different implementations: a *server-based* and a *client-based* one. The decentralized server-based implementation consists of multiple server-based caches (i.e. as many caches as FVPD partitions). Here, improvements are expected, as query-matching rows are directly collected from fast cache primary storage devices rather than from comparatively slower database storages (HDD or SSD). Table 1.2 illustrates all approaches that are used to test this hypothesis.

Table 1.2: Approaches for Hypothesis 2

| **Approach** | **Used Name in Thesis** |
|---|---|
| Decentralized Server-Based Caching | Server-Based Caching |
| Local Decentralized Client-Based Caching | Local Caching |
| Remote Centralized Server-Based Caching | Remote Caching |

Besides that decentralized server-based approach, there are two other implementations: a decentralized client-based cache, denoted as *local* cache, where a cache resides directly at the client and a *remote* cache, where the cache is a dedicated server between the FVPD partitions and all clients. In the *local* one, the cache is located directly on the client that issues the database query. Hence, this *local* cache stores entirely reconstructed rows, performance improvements are expected. Here, it is viable to cache completely reconstructed rows, as similar to all other approaches the client is the final place where the rows are reconstructed eventually.

Yet, a single centralized *remote* cache, which also stores entirely reconstructed rows contradicts the *Security-by-Distribution* principle of *SeDiCo*. Nevertheless,

this approach is also evaluated in case further security aspects to protect this centralized cache are applied[21].

There are also two further analogous cache memory implementations: a file-based JSON and a local relational database used a cache, evaluated and compared. These implementations are also evaluated for the sake of comparability against the In-Memory cache.

The investigation of this hypothesis will show to which extent the response time can be improved and how this approach performs compared against the initial *SeDiCo* implementation and against the other approaches.

*Hypothesis 3: Using Solid State Disks (SSDs) as distributed secondary storage devices for the FVPD data improves the response time to a level that is in the same order of magnitude as a non-partitioned and non-distributed scenario due to faster access times of the memory.*

The investigations concerning this hypothesis are empirical measurements. Here, no new query strategies or algorithms are developed, but the original *SeDiCo* framework is operated as is with faster hardware capabilities (i.e. SSDs, Solid State Drives). Thus, the impact of using new technology is measured and the results will show if the response time can be improved as expected.

The basic assumption is that due to the usage of SSD devices as secondary storage for the FVPD data the response time is improved. SSDs store data in so-called flash memory, which is non-volatile and can be accessed electronically by a storage controller. In contrast to this, a traditional HDD (hard disk drive) consists of rotating magnetic disks, where so-called heads placed above these disks, read and write data. Hence, moving the head (mechanically) over the disks is slower compared to the (completely electronic) storage controller of SSDs. To evaluate the response time of this approach, no changes of the original *SeDiCo* framework are made, except that the FVPD data are stored in databases that use SSDs as secondary storage. Thus, this approach contains a practical performance measurement, with no algorithmic considerations. Here, the influence of a new hardware generation is investigated.

The basic goal of this work is to find *query strategies* that work on FVPD data in the same order of magnitude as on non-partitioned and non-distributed environments. Therefore, the afore-mentioned hypotheses are analyzed with 3

[21] e.g. place the cache inside a separate network, only accessible via virtual private network

corresponding execution methods: a *query rewriting*, a *caching*, and an *SSD-based* one.

Table 1.3 assigns the analyzed approaches to their corresponding hypothesis in order to provide a general overview.

Table 1.3: Thesis Approaches Mapped to Hypotheses

| **Approach** | **Hypothesis** | **Execution Method** |
|---|---|---|
| Query Rewriting | Hypothesis 1 | Query Rewriting |
| Server-Based Caching | Hypothesis 2 | Caching |
| Local Caching | Hypothesis 2 | Caching |
| Remote Caching | Hypothesis 2 | Caching |
| SSD-based Approach | Hypothesis 3 | SSD-based |

During the course of this thesis all approaches will be conceptualized, implemented, and evaluated. Moreover, the evaluation is distinguished in two phases: a *local* and a *remote* one.

### Local and Remote Evaluation

In the *local* evaluation all components are installed on one single machine. The main reason for this is to avoid typical side-effects in Cloud Computing environments (i.e. changing (Internet) network bandwidths, unknown utilization of physical hosts and virtual machines, etc.). In contrast to the *local* evaluation, the *remote* evaluation uses a (private) cloud infrastructure and a client connected via local area network (LAN) to reduce the above-mentioned side-effects to a minimum.

# Chapter 2

# Definition of the FVPD Methodology and its Original Implementation in the *SeDiCo* Framework

This chapter introduces a formal definition of the FVPD methodology and proves its correctness. It also illustrates the complete *SeDiCo* framework development[1] that this thesis is based on, and creates a common understanding of it.

## 2.1 Fixed Vertical Partitioning and Distribution (FVPD) Definition

This section defines the key elements of the FVPD methodology. The main idea is that a *relation* is divided in several *partitions* in a way, such that each individual *partition* contains logically independent *tuples*. Thus, in order to use FVPD data, they have to be *joined* first and this requires mechanisms to separate a *relation* in several parts and strategies to query the respective *partitions*, such that the *join* produces an equal result set compared to the original query over the original *relation*. In the rest of the thesis it is assumed (without the loss of generality) that the original *relation* contains one single *primary key attribute* and that its

[1] that was conducted by the author in cooperation with student works listed at the end of the thesis from 2012 to 2014

FVPD *partitions* as *two relations* satisfy the necessary and sufficient conditions to represent the original *relation*. The presented results of the thesis are also correct for *relations* with more than one *primary key attribute* and more than *two FVPD partitions*, as the following sections show.

### 2.1.1 Vertical Data Partitioning

In this section, necessary notions for the definition of the FVPD methodology are presented. Furthermore, the correctness of the FVPD methodology is proved. For this, firstly a non-FVPD relation and then the corresponding FVPD relations are defined.

**Definition 1.** *Non-FVPD relation*

*Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes. Let $R(A)$ be a relation $R$ with attributes $A$ such that $a_1$ is the only key attribute for $R(A)$. Then the relation $R$ is called a* **Non-FVPD relation**. □

In other (more informal) words, any relation $R$ that is a database table with attributes $A$, with a single *primary key attribute* $a_1$, is a **Non-FVPD relation**. This is a relation in the original database which will be vertically partitioned and distributed to different clouds in order to protect the data (*tuples*) from impermissible access.

**Definition 2.** *FVPD relations for the non-FVPD relation $R(A)$*

*Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and let $R$ be a non-FVPD relation with attributes $A$. Let $B$ and $C$ be two sets of attributes such that:*

- $B \cup C = A$,

  *and*

- $B \cap C = \{a_1\}$

*Then, the two relations $S_v(B)$ and $T_v(C)$ are* **FVPD relations** *for the non-FVPD relation $R(A)$, if and only if*

- $|S_v(B)| = |T_v(C)| = |R(A)|$

  *and*

- $R(A) = S_v(B) \bowtie_{a_1} T_v(C)$. □

The condition $B \cap C = \{a_1\}$ is called **disjointness** criterion, because the sets of attributes in the partitions $B$ and $C$ are disjoint except for the primary key attribute $a_1$. The condition $|S_v(B)| = |T_v(C)| = |R(A)|$ is called **completeness** criterion, because there are one-to-one correspondences from sets of tuples in $S_v(B)$ to the set of tuples in $T_v(C)$ on the basis of the value of the primary key attribute $a_1$. The completeness criterion ensures that in $S_v(B)$ and $T_v(C)$ there is no tuple with a value for the primary key attribute for which there is no tuple in $R(A)$ with the same value for the primary key attribute.

In other words, relation $R(A)$ is vertically partitioned into 2 *disjoint* relations $S_v(T)$ and $T_v(C)$, except for the primary key attribute $a_1$ which is replicated to both FVPD relations to reconstruct (i.e. join) the original non-FVPD relation. Because of this join, the number of rows (i.e. the cardinality) in all relations must be equal (*completeness*).

The definition for more than two FVPD relations works analogously and is not pointed out any further for the sake of a better readability. Similarly, FVPD partitions with more than one primary key attribute are possible, but not presented, since here, not only one key attribute ($a_1$), but a set of key attributes must be replicated in all FVPD partitions.

**Definition 3.** *Reconstruction queries*

*Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and let $R$ be a non-FVPD relation with attributes $A$. Let $S_v(B)$ and $T_v(C)$ be FVPD relations for relation $R(A)$.*

*Let $\Pi_{(a_i,\ldots,a_j)}, (1 \leq i < j \leq n)$ be a projection query for $R(A)$, such that*

$RS \leftarrow \Pi_{(a_i,\ldots,a_j)} R(A)$.

*Let $\Pi_{v_1(a_1,a_k,\ldots,a_l)}$ be a projection query for $S_v(B)$ and let $\Pi_{v_2(a_1,a_m,\ldots,a_o)}$ be a projection query for $T_v(C)$ with $1 \leq i \leq k, l, m, o \leq j \leq n$, such that*

$RS_{v1} \leftarrow \Pi_{v_1(a_1,a_k,\ldots,a_l)} S_v(B)$ *and* $RS_{v2} \leftarrow \Pi_{v_2(a_1,a_m,\ldots,a_o)} T_v(C)$.

*The projections queries $\Pi_{v_1(a_1,a_k,\ldots,a_l)}$ and $\Pi_{v_2(a_1,a_m,\ldots,a_o)}$ are called* **reconstruction queries** *for the projection query $\Pi_{(a_1,\ldots,a_j)}$, if and only if*

$RS = \Pi_{(a_i,\ldots,a_j)}(RS_{v1} \bowtie_{a_1} RS_{v2})$. □

In other words the two projection queries $\Pi_{v_1(a_1,a_k,\ldots,a_l)}$ and $\Pi_{v_2(a_1,a_m,\ldots,a_o)}$ are reconstruction queries for the original projection query $\Pi_{(a_1,\ldots,a_i)}$ if the result sets $RS_{v1}$ and $RS_{v2}$ can be joined to the result set $RS$ of the original query $\Pi_{(a_i,\ldots,a_j)}$. The additional projection over the join of $RS_{v1}$ and $RS_{v2}$ is necessary because the primary key attribute $a_1$, which is crucial for the join can be left out in the attribute list of the original query: $(a_i, \ldots, a_j)$.

**Definition 4.** *FVPD methodology*

*Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and let $R$ be a non-FVPD relation with attributes $A$. Let $B = \{a_1, a_2, \ldots, a_k\}$ and $C = \{a_1, a_{k+1}, \ldots, a_n\}$ for $2 \leq k \leq n-1$ be two sets of attributes such that:*

- $B \cup C = A$,

  *and*

- $B \cap C = \{a_1\}$,

  *and*

- *the result sets for the projections on $B$ and $C$: $RS_{v1} \leftarrow \Pi_{(a_1,a_2,\ldots,a_k)}R(A)$ and $RS_{v2} \leftarrow \Pi_{(a_1,a_{k+1},\ldots,a_n)}R(A)$ contain no sensitive or relevant information respectively.*

*Then, the two relations $S_v(B) = RS_{v1}$ and $T_v(C) = RS_{v2}$ are* **FVPD relations** *for the non-FVPD relation $R(A)$.*

*In addition to this, the FVPD methodology stores the two FVPD relations in two different locations (i.e. clouds) which improves the level of security and privacy for the respective data.* □

With respect to this definition, it is important to note that the data (tuples) in just one of the FVPD partitions do not contain any security or privacy relevant data. This is something that has to be determined by the framework user who knows about security and privacy issues related to the actual data. If it is not possible to partition data in two FVPD partitions, then the partitioning in three or more relations is possible analogously, but this is out of the scope of this thesis.

In very rare cases, when each column by itself contains sensitive data with respect to security and privacy, then this methodology is not applicable.

Additionally, in the definition for the sake of better readability it is assumed that the attributes within the sets $B$ and $C$ are *ordered* as in the set $A$. However, this is not a restriction because the attributes in $A$ can always be reordered such that they correspond to the ordering in the FVPD partitions.

Having defined the FVPD methodology now, the correctness proof has to show that the FVPD partitions represent the same data as the original relation. This is done by showing that for each query for the original relation, there exist *reconstruction queries*, and this is done in the next section.

### 2.1.2 Correctness of FVPD methodology

Theorem 1[2] states the correctness of the original *SeDiCo* approach. The proof of the theorem consists in two steps: (1) presentation of the algorithm for the query rewriting into the reconstruction queries, and (2) the proof that the two rewritten queries are in fact *reconstruction queries* for the original one.

**Theorem 1.** *Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and let $R$ be a non-FVPD relation with attributes $A$. Let $S_v(B)$ and $T_v(C)$ be FVPD relations for relation $R(A)$.*

*For each $\Pi_\omega$, $(\omega = (a_i, \ldots, a_j) : 1 \leq i < j \leq n)$, projection query for $R(A)$, such that*

$RS \leftarrow \Pi_\omega R(A)$,

*there exist two projection queries $\Pi_{v_1(a_1,a_k,\ldots,a_l)}$ for $S_v(B)$ and $\Pi_{v_2(a_1,a_m,\ldots,a_o)}$ for $T_v(C)$, that are reconstruction queries for the original projection query $\Pi_\omega$[3].*

**Proof for Theorem 1.** The basic idea behind this correctness proof is to show that a result set based on a *projection query* against a *non-FVPD relation* is equal to the result set based on the join of two result sets of the corresponding *reconstruction queries* against their *FVPD relations*. The following proof shows that for each projection query there exist two *reconstruction queries*.

[2]For the sake of better readability, this proof is illustrated with a projection query $\Pi$, and two FVPD partitions but works analogously for all other query types, their combinations, and for more than two FVPD partitions.

[3]note that if the primary key attribute $a_1$ is not contained in the original query, it is automatically added by the framework based on the relation's metadata. This ensures that the result set does not contain duplicate rows.

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of attributes and let $R$ be a non-FVPD relation with attributes $A$. Let $S_v(B)$ and $T_v(C)$ be FVPD relations for relation $R(A)$. Thus, for $B$ and $C$ it holds:

- $B \cup C = A$,

  and

- $B \cap C = \{a_1\}$,

Let $\Pi_\omega$, $(\omega = (a_i, \ldots, a_j) : 1 \leq i < j \leq n)$, be a projection query for $R(A)$ and $RS \leftarrow \Pi_\omega R(A)$.

Then, two projection queries $\Pi_{v_1\omega_1}$ for $S_v(B)$ and $\Pi_{v_2\omega_2}$ for $T_v(C)$ are constructed. The following algorithm proves that they are reconstruction queries for the query $\Pi_\omega$:

**Data:**
$B$; /* the set of attributes for partition $S_v(B)$ */
$C$; /* the set of attributes for partition $T_v(C)$ */
$\omega = \{a_i, \ldots, a_j\}$; /* the set of query restrictions */
**Result:**
$\Pi_{v_1\omega_1}$ and $\Pi_{v_2\omega_2}$; /* the reconstruction queries */
**begin**
  $\omega_1 \leftarrow \{a_1\}$; /* the set of restriction for query $\Pi_{v_1\omega_1}$ */
  $\omega_2 \leftarrow \{a_1\}$; /* the set of restriction for query $\Pi_{v_2\omega_2}$ */
  **foreach** $a \in \omega \setminus \{a_1\}$ **do**
    **if** $a \in B$ **then**
      $\omega_1 \leftarrow \omega_1 \cup \{a\}$;
    **else**
      $\omega_2 \leftarrow \omega_2 \cup \{a\}$;
    **end**
  **end**
  **return** $\Pi_{v_1\omega_1}$, $\Pi_{v_2\omega_2}$
**end**

**Algorithm 1:** The Algorithm For The Creation Of The *Reconstruction Queries*

Algorithm 1 produces in a finite number of steps (as there are a finite number of attributes) two sets of attributes $\omega_1$ and $\omega_2$ and on the basis of them it constructs

two projection queries $\Pi_{v_1\omega_1}$ for $S_v(B)$ and $\Pi_{v_2\omega_2}$ for $T_v(C)$. These queries produce the result sets $RS_{v1} \leftarrow \Pi_{v_1\omega_1}S_v(B)$ and $RS_{v2} \leftarrow \Pi_{v_2\omega_2}T_v(C)$ respectively. The rest of the proof shows that they are *reconstruction queries* for the query $\Pi_\omega$.

Let $\Pi_\omega$, $(\omega = (a_i, \ldots, a_j) : 1 \leq i < j \leq n)$, be a projection query for $R(A)$ and let $\Pi_{v_1\omega_1}$ and $\Pi_{v_2\omega_2}$ be two projection queries constructed by Algorithm 1.

Let $RS \leftarrow \Pi_\omega R(A)$ and $RS' \leftarrow \Pi_\omega(RS_{v1} \bowtie_{a_1} RS_{v2})$ be result sets. The goal of the proof is to show that $RS = RS'$.

**1. Implication ($\Rightarrow$):**

Let $t_l^\omega = (r_{a_i,l}, ..., r_{a_j,l})$ where $(r_{a_i,l}, ..., r_{a_j,l}) \in RS$.

Then there is a row $t_m^A = (r_{a_1,m}, ..., r_{a_n,m})$ where $(r_{a_1,m}, ..., r_{a_n,m}) \in R(A)$ for some $m$ such that

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l^\omega \wedge r_{a,m} \in t_m^A) \rightarrow r_{a,l} = r_{a,m})$$

This follows from the definition of projection query: for each row in the result set there is a row in the relation, from which the row in the result set is projected. This is denoted as $t_l^\omega = \Pi_\omega t_m^A$ - applying the projection query on a single row in the relation $R(A)$. Note that by the definition of a *projection* (Codd, 1970), duplicates are not contained in the result set $RS$, and as the primary key attribute $a_1$ is always part of the reconstruction queries $\Pi_{v1}$ and $\Pi_{v2}$, it is also assured that $RS'$ does not contain any duplicates.

For $t_m^A$ there are two rows $t_m^B = (r_{a_1,m}, ..., r_{a_q,m})$ and $t_m^C = (r_{a_1,m}, ..., r_{a_p,m})$ where $(r_{a_1,m}, ..., r_{a_q,m}) \in S_v(B)$ and $(r_{a_1,m}, ..., r_{a_q,m}) \in T_v(C)$ such that

$$t_m^A = t_m^B \oplus_{a_1} t_m^C$$

This follows from Definition 2 of FVPD relations. $\oplus_{a_1}$ is the concatenation of of the two rows on the basis of the values of the key attribute $a_1$ which have to be the same in both rows and this value is presented just once in the result.

Let $t_m^{\omega_1} = \Pi_{v_1\omega_1} t_m^B$ and $t_m^{\omega_2} = \Pi_{v_2\omega_2} t_m^C$ be the projection queries constructed by Algorithm 1. Then, it follows from Algorithm 1 and Definition 2 of the FVPD relations:

$$\forall a.((a \in \omega \wedge a \in B \wedge r_{a,l} \in t_l^{\omega}) \rightarrow r_{a,l} \in t_m^{\omega_1})$$

and

$$\forall a.((a \in \omega \wedge a \in C \wedge r_{a,l} \in t_l^{\omega}) \rightarrow r_{a,l} \in t_m^{\omega_2})$$

Therefore

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l^{\omega}) \rightarrow r_{a,l} \in (t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2}))$$

then it follows

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l^{\omega}) \rightarrow r_{a,l} \in \Pi_{\omega}(t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2}))$$

In the other direction, it follows that

$$\forall a.((a \in \omega_1 \wedge a \in \omega \wedge r_{a,m} \in t_m^{\omega_1}) \rightarrow r_{a,m} \in t_l^{\omega})$$

and

$$\forall a.((a \in \omega_2 \wedge a \in \omega \wedge r_{a,m} \in t_m^{\omega_2}) \rightarrow r_{a,m} \in t_2^{\omega})$$

Therefore

$$\forall a.((a \in (\omega_1 \cup \omega_2) \wedge a \in \omega \wedge r_{a,m} \in (t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2})) \rightarrow r_{a,m} \in t_l^{\omega})$$

from this it follows

$$\forall a.((a \in \omega \wedge r_{a,m} \in \Pi_{\omega}(t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2})) \rightarrow r_{a,m} \in t_l^{\omega})$$

.

Therefore for each $t_l^{\omega} \in RS$ there is a $t'^{\omega} \in \Pi_{\omega}(RS_{v1} \bowtie_{a_1} RS_{v2})$ such that

$$t_l^{\omega} = t'^{\omega}$$

From this follows that

$$RS \subseteq RS'. \tag{2.1}$$

**2. Implication ($\Leftarrow$):**

Let $t_l'^{\omega} = (r_{a_i,l}, ..., r_{a_j,l})$ where $(r_{a_i,l}, ..., r_{a_j,l}) \in RS'$. Which means that $t_l'^{\omega} \in \Pi_{\omega}(RS_{v1} \bowtie_{a_1} RS_{v2})$.

Then there is a row $t_m^{\omega_1,\omega_2} = (r_{a_1,m}, ..., r_{a_k,m}), (1 \leq k \leq n)$, where $(r_{a_1,m}, ..., r_{a_k,m}) \in (RS_{v1} \bowtie_{a_1} RS_{v2})$ for some $m$ such that

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l'^{\omega} \wedge r_{a,m} \in t_m^{\omega_1,\omega_2}) \rightarrow r_{a,l} = r_{a,m})$$

Note that by the definition of a *projection* (Codd, 1970), duplicates are not contained in the result set $RS$, and as the primary key attribute $a_1$ is always part of the reconstruction queries $\Pi_{v1}$ and $\Pi_{v2}$, it is also assured that $RS'$ does not contain any duplicates.

For $t_m^{\omega_1,\omega_2}$ there are two rows $t_m^{\omega_1} = (r_{a_1,m}, ..., r_{a_q,m})$ and $t_m^{\omega_2} = (r_{a_1,m}, ..., r_{a_p,m})$ where $(r_{a_1,m}, ..., r_{a_q,m}) \in RS_{v1}$ and $(r_{a_1,m}, ..., r_{a_p,m}) \in RS_{v2}$ such that

$$t_m^{\omega_1,\omega_2} = t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2}$$

This follows from the definition of the *join*. The two rows $t_m^{\omega_1}$ and $t_m^{\omega_2}$ share the same value $r_{a_1,m}$ for the key attribute $a_1$. Above that, there are two rows $t_m^B \in S_v(B)$ and $t_m^C \in T_v(C)$ such that $t_m^{\omega_1} = \Pi_{v_1\omega_1} t_m^B$ and $t_m^{\omega_2} = \Pi_{v_2\omega_2} t_m^C$. This follows from the definition of the two *projection queries* created by Algorithm 1. The two rows $t_m^B$ and $t_m^C$ also share the same value $r_{a_1,m}$ for the key attribute $a_1$ which also follows from the definition of the *projection queries* created by Algorithm 1. Therefore, there is a row $t_m^A \in R(A)$ such that $t_m^A = (t_m^B \oplus_{a_1} t_m^C)$. This follows from Definition 2 of FVPD relations.

From all this, it follows

$$\forall a.((a \in \omega \wedge a \in B \wedge r_{a,l} \in t_l'^{\omega}) \rightarrow r_{a,l} \in t_m^B)$$

and

$$\forall a.((a \in \omega \wedge a \in C \wedge r_{a,l} \in t_l'^{\omega}) \rightarrow r_{a,l} \in t_m^C)$$

from Algorithm 1. Therefore,

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l'^{\omega}) \rightarrow r_{a,l} \in (t_m^B \oplus_{a_1} t_m^C))$$

then it follows

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l'^{\omega}) \rightarrow r_{a,l} \in \Pi_{\omega}(t_m^B \oplus_{a_1} t_m^C))$$

then

$$\forall a.((a \in \omega \wedge r_{a,l} \in t_l'^{\omega}) \rightarrow r_{a,l} \in \Pi_{\omega}(t_m^A))$$

Let $t_m^{\omega} = \Pi_{\omega}(t_m^A)$. In the other direction, it follows that

$$\forall a.((a \in \omega_1 \wedge a \in \omega \wedge r_{a,m} \in t_m^{\omega}) \rightarrow r_{a,m} \in t_m^{\omega_1})$$

and

$$\forall a.((a \in \omega_2 \wedge a \in \omega \wedge r_{a,m} \in t_m^{\omega}) \rightarrow r_{a,m} \in t_m^{\omega_2})$$

Therefore

$$\forall a.((a \in (\omega_1 \cup \omega_2) \wedge a \in \omega \wedge r_{a,m} \in t_l^{\omega}) \rightarrow r_{a,m} \in (t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2}))$$

from this, it follows

$$\forall a.((a \in \omega \wedge r_{a,m} \in t_l^{\omega}) \rightarrow r_{a,m} \in \Pi_{\omega}(t_m^{\omega_1} \oplus_{a_1} t_m^{\omega_2}))$$

.

Therefore, for each $t_l'^{\omega} \in \Pi_{\omega}(RS_{v1} \bowtie_{a_1} RS_{v2})$ there is $t_m^{\omega} \in RS$ such that

$$t_l'^{\omega} = t_m^{\omega}$$

From this follows that

$$RS' \subseteq RS \tag{2.2}$$

From 2.1 and 2.2 it follows that

$$RS = RS'$$

This proves the theorem. □

Theorem 1 shows that for each projection query with respect to the non-FVDP relation, there exist two *reconstruction queries* over the corresponding FVPD relations. The primary key attribute $a_1$ needed a special treatment in Algorithm 1 because the original projection query $\Pi_{(a_i,\ldots,a_j)}$ can be not restricted with respect to $a_1$: the key attribute is necessary for the join of the result sets from the two *reconstruction queries*. In the algorithm, it is therefore added to both sets of attributes $\omega_1$ and $\omega_2$. The final projection on the original set $\omega$ ensures that the column corresponding to the key attribute is deleted from the result set if it is not contained in $\omega$.

This proof verifies hypothesis 0, stating that the FVPD methodology improves the level of security and privacy in the context of relational databases. As stated in Definition 4, the approach can be extended to more than *two FVPD partitions* and to more than *one primary key attribute*. Here *SeDiCo*, as an implementation of this methodology, not only provides a framework but also showed the technological feasibility. Nevertheless, it has to be noted that the definition of a secure FVPD data scheme is in the responsibility of the framework user and that it is not applicable if a single column of a relation contains security or privacy-relevant data.

## 2.2 Data Distribution: The *SeDiCo* Approach

Having outlined the basic principles of the FVPD methodology and its proof, this section gives an overview about the original implementation of the methodology in form of the *SeDiCo* framework.

The basic approach of *SeDiCo* (A *SE*cure and *DI*stributed *C*loud Data st*O*re) is to divide data into several partitions and distribute them across various clouds. Thus, every cloud provider only gets a chunk of the data that is worthless without the other parts. Based on this logical and physical data distribution, the level of security and privacy in the cloud is enhanced.

Figure 2.1 illustrates the *Security-by-Distribution* approach with a simplified example based on the TPC-W *CUSTOMER* relation.

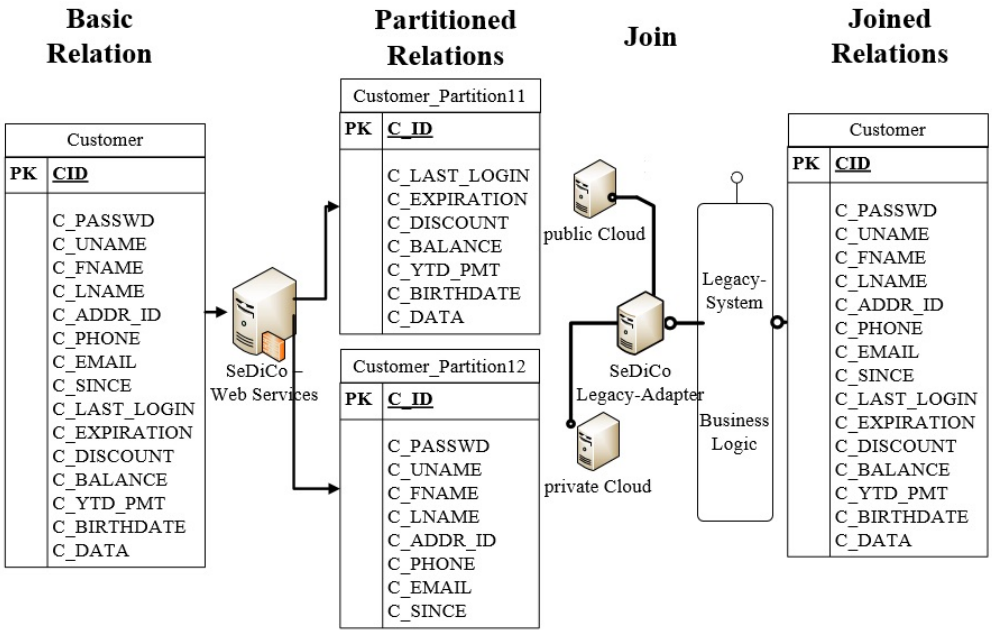



Figure 2.1: *SeDiCo* Architecture with TPC-W CUSTOMER Data Scheme

Since it is possible to distribute data across various clouds and various database systems, the entire setup can be regarded as a so-called *distributed database system* (Elmasri & Navathe, 2015). Here, (Elmasri & Navathe, 2015) distinguish between *multi-database* (with no global data scheme) and *federated* (with a global data scheme) database systems. In a so-called multi-database system, the clients create their required schemes on demand, whereas in a federated database they rely on a global scheme that unites all involved distributed schemes. However, (Elmasri & Navathe, 2015) also state that a clear distinction between those kinds of database systems cannot be made and both types are often subsumed under

the notion of a federated database system. According to this, the entire *SeDiCo* approach is also regarded as a federated database architecture[4], because of the initial non-partitioned and non-distributed relation.

As data are stored across different clouds, no provider has access to entire rows. To stick with Figure 2.1, if a public cloud provider is attacked and *Customer_Partition11* is stolen, the data is useless for the attacker without the corresponding *Customer_Partition12*. In order to realize such a concept, adequate cloud infrastructures are a crucial point. *SeDiCo* showed the technical feasibility of this idea with a Java-based implementation. Here, two different clouds (Amazon EC2 (Amazon, 2016) as a public and Eucalyptus (Hewlett Packard, 2016) as a private cloud) and two different databases (MySQL (MySQL, 2016) and Oracle Express (Oracle, 2016)) were used. The mapping of different database types is done via Hibernate (cf. Section 3.3.2) and different cloud APIs are encapsulated with jclouds (Apache, 2016b).

A concluding architectural overview about all these concepts can be found in Figure 2.2.

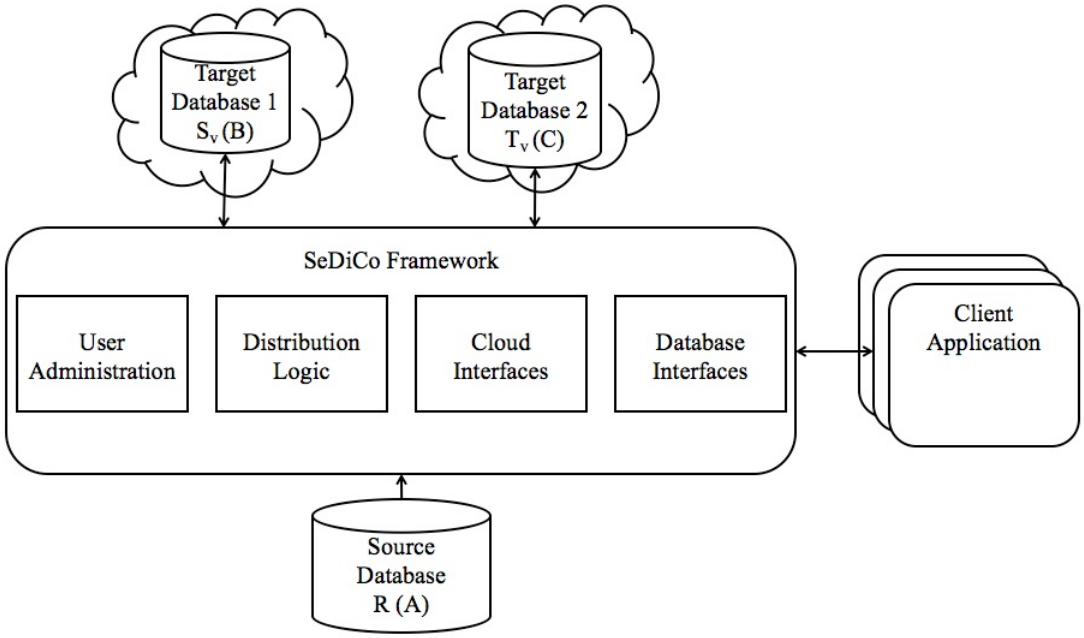



Figure 2.2: *SeDiCo*'s Architectural Overview

The *SeDiCo* framework targets on database administrators, developers and architects, who aim at transferring database data into a dynamically scalable cloud infrastructure. Also, system administrators and architects who intend to use

[4]even if just one (distributed) database system (e.g. MySQL) is used

a cloud-based infrastructure for creating redundant or high availability database systems are addressed. For these target groups, *SeDiCo* offers a solution to use all kinds of cloud deployment models, i.e. public, private, hybrid and community clouds, for the storage of database data, which is transparently usable for new but also legacy applications.

Figure 2.2 illustrates the entire *SeDiCo* framework. *SeDiCo* is implemented in Java as the most widely used programming language in nowadays enterprises (TIOBE, 2016). Basically, there are four central aspects: the *user administration*, the *distribution logic*, the *cloud interfaces* and the *database interfaces*. The key components for this thesis are the *distribution logic* and the *database interfaces*. It is possible to use the *SeDiCo* framework with different database implementations (e.g. MySQL, Oracle, MariaDB, etc.). However, although these database systems implement the SQL standard, the concrete implementation differs from database system to database system. This requires an additional layer that abstracts from the concrete database system implementations and this is done in the *database interfaces* component with Hibernate (RedHat, 2016) as an ORM (Object-Relational Mapper)[5]. Here, Hibernate introduces a high-level query language (JPQL, Java Persistence Query Language) upon SQL, which is independent from the concrete underlying database system. Hibernate and its implications for *SeDiCo* is introduced in more detail in Section 3.3 and the distribution logic component of *SeDiCo* is outlined in the following sections.

### 2.2.1 FVPD Join

A key element in *SeDiCo* is the *join* of rows that match a query. Transferred to the presented FVPD approach, a *join* corresponds to *joining* query matching rows in order to reconstruct them. Thus, all join algorithms that are described in this section implement the *natural join* (cf. Section 1.2.3), and the replicated *primary key attribute* $a_1$ appears only once in the respective result set. Thus, for the join both partitions $S$ and $T$ has to be iterated to find query-matching rows. This results in a run time complexity (with respect to the response time) of $\mathcal{O}(n^2)$, with $n$ indicating the cardinality of $S$ and $T$.

[5]An ORM has several advantages: firstly, it bridges the gap between the object-oriented programming and the relational database paradigms, secondly, it abstracts from a concrete database implementation, and thirdly, it ensures transaction safety with the usage of a so-called *session*, cf. Section 3.3

The complexity of this initial FVPD approach (without any optimization) is summarized in Table 2.1.

Table 2.1: Query Mechanism Complexity

| Query Mechanism | Join Algorithm | Complexity |
|---|---|---|
| Initial FVPD approach | Nested-Loops Join | $\mathcal{O}(n^2)$ |

### 2.2.2 Row Reconstruction in *SeDiCo*

The row reconstruction in *SeDiCo* is performed in 2 steps: a *row collection* and a *join* step. These steps are illustrated in Figure 2.3.

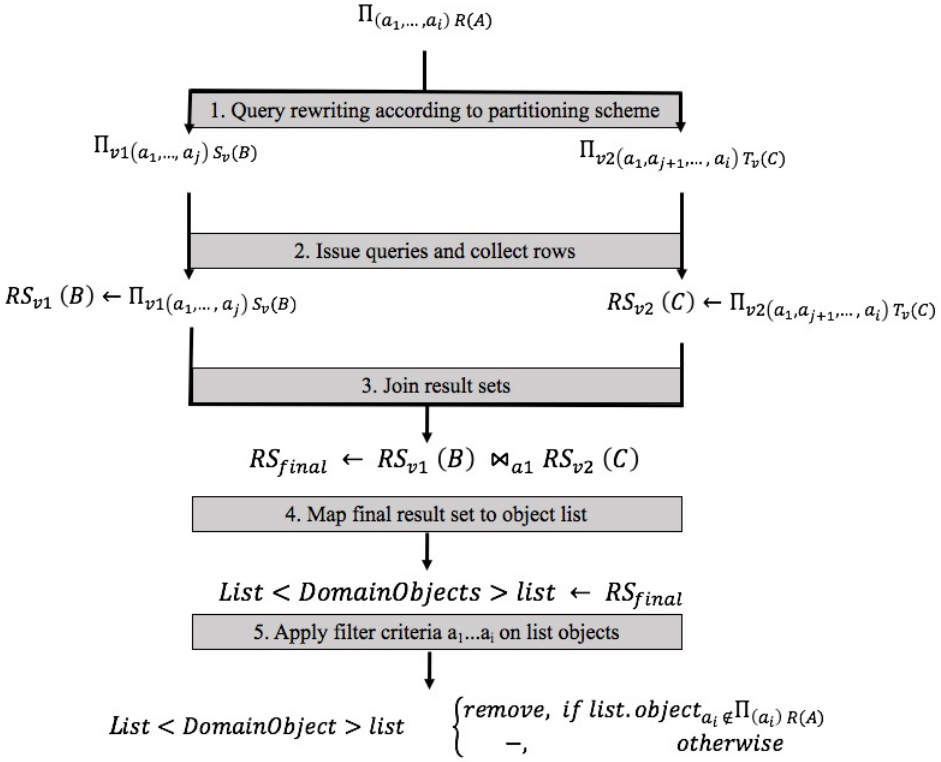



Figure 2.3: Query and Join Approach in *SeDiCo*

1. Row Collection

   First and foremost, the original query $\Pi_{(a_1,\dots,a_i)}R(A)$ is analyzed with respect to the partitioning scheme, which is defined by the framework user in an XML file[6]. The query attributes $(a_1, \dots, a_i)$ are assigned to the respective

[6]the *primary key attribute* $a_1$ is also specified in the XML file, so if the original query does not contain the *primary key attribute*, it is automatically added by the *SeDiCo* framework

partitions and the original query is rewritten into so-called *reconstruction queries* for the corresponding vertical partitions:

$$\Pi_{v1(a_1,\ldots,a_j)} S_v(B)$$
$$\Pi_{v2(a_1,a_{j+1},\ldots,a_i)} T_v(C)$$

with $B \subseteq A$, $C \subseteq A$, $B \cap C = \{a_1\}$, $B \cup C = A$, $a_j$ as the $j$th attribute from A, $j < i$, and $i$ as the number of attributes in $A$.

Subsequently, these *reconstruction queries* are issued against the respective partitions and their result sets are collected accordingly:

$$RS_{v1} \leftarrow \Pi_{v1(a_1,\ldots,a_j)} S_v(B)$$
$$RS_{v2} \leftarrow \Pi_{v2(a_1,a_{j+1},\ldots,a_i)} T_v(C)$$

The rows in these result sets are then joined with a *natural join* on their *primary key* into the final result set $RS_{final}$:

$$RS_{final} \leftarrow RS_{v1} \bowtie_{a_1} RS_{v2}$$

Now, action 4 (Figure 2.3) writes the result set $RS_{final}$ (relational paradigm) to a list of objects $List < DomainObject > list$ (object-oriented paradigm). Here, Hibernate as the Object-Relational Mapper (ORM) is necessary to be independent from the underlying database system. This object representation of the result set abstracts from a concrete database implementation and enables the framework user to use various systems.

This finishes the first step and if there are filter criteria $\omega_i$, the second step has to be performed.

2. Row Filtering

   Now, all rows are entirely reconstructed as objects, and filter criteria $\omega_i$ can be applied on the respective attributes. This facilitates the entire row

reconstruction step[7], as especially the query attribute concatenation with logical ANDs and ORs is a complex task to distribute into the respective queries. Therefore, a list of domain objects (mapping the rows to objects) is built:

$$List < DomainObject > list = RS_{final}$$

Then, (if there exist) query attributes $\omega$ are applied with their respective combinations (e.g. *AND*, *OR*, *NOT*, etc.) on the objects in this list. If they match, the respective object remains in the list and if they do not match, the respective object is removed from the list:

$$List < DomainObject > list \begin{cases} \text{remove, if } list.object_{a_i} \notin \Pi_{(a_i)} \\ \text{-, otherwise} \end{cases}$$

In the end, this list contains the final result set of the original query, represented as a list of objects.

### 2.2.3 FVPD CRUD Operations

In order to provide an exhaustive overview about the entire *SeDiCo* approach, this section now outlines the implementation of the 4 basic database operations: create, read, updated, and delete (CRUD), based on the FVPD approach.

Figure 2.4 illustrates the implementation of the CRUD operations, based on Hibernates *Interceptor* mechanism which uses so-called *listeners* (RedHat, 2016). The application logic (here in form of a Java client) uses the ORM which maps the objects to the corresponding relations. As soon as an object is accessed[8] (within a Hibernate session), a listener intercepts this object call and translates the original SQL statement into an SQL statement that corresponds to the FVPD scheme. An example of such a translation can be found in Figure 2.3. After executing the translated SQL statements against the respective partitions, the ORM is used to collect the result sets and reconstructs the rows with joining all query-matching

[7] and the correctness proof in Section 2.1.2
[8] i.e. created, read, updated, or deleted

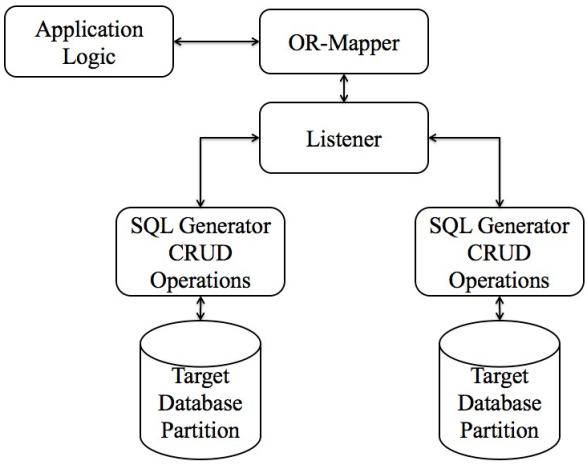



Figure 2.4: CRUD Operations in *SeDiCo*

rows on their primary keys. This procedure is analogous for all CRUD operations, i.e. create (persisting an object to the partitions), read (querying for one or more rows), update (persisting changes of an object into the partitions) and delete (removing an object and deleting it from the partitions). Considering the fact that this approach involves several databases (respective partitions), ideally distributed across different cloud vendors, another challenging task is to preserve the ACID (atomicity, consistency, isolation, and durability) criteria in such a distributed environment. The next section concentrates on the maintenance of the ACID criteria.

### ACID Criteria in *SeDiCo*

Since the entire approach relies on relational but vertically partitioned data, the question of how to ensure all ACID criteria emerges. Using Hibernate as ORM means, that all CRUD operations must be performed within a *session* in the Java client (RedHat, 2016). Such a (Java) *session* object is then used to create a transaction around the respective CRUD operation and this transaction ensures all ACID criteria, even in such a FVPD environment. This preserves the so-called *hard consistency* throughout the entire *SeDiCo* framework:

- *A*tomicity: a transaction (considered as a basic set of database operations) is either performed entirely (commit) or not at all (rollback)
- *C*onsistency: the database system is in a consistent state before and after each transaction

- *I*solation: transactions are independent from each other and cannot access data that is simultaneously processed by another one
- *D*urability: data is long-lastingly stored in a database

The presented version of *SeDiCo* was developed and implemented before the work on the thesis has started. The results of this preliminary work have shown that the ideas behind *SeDiCo* are feasible and work in practice. However, there is still an open question regarding the framework performance in practical use cases:

- **Performance optimization of the *SeDiCo* approach:** Although the feasibility of the original implementation is empirically shown and formally proved, the response time (especially for larger data sets, i.e. more than 10K rows) decreased tremendously. Thus, how can the response time for a FVPD query in practical use cases scenarios be improved, such that it is in the same order of magnitude as a non-FVPD query?

This section closes the presentation of the entire *SeDiCo* framework and created a common understanding of how the approach works. The following related work with respect to *SeDiCo* embeds it into a broader state-of-the-art context. Then, the subsequent chapters conceptualize, implement and evaluate *optimized query mechanisms* for the *SeDiCo* framework.

# Chapter 3

# Related Work

This chapter covers the architectural background for the key concepts of the entire *SeDiCo* framework[1] and relates them to current research topics. The structure of this chapter is aligned to Figure 3.1[2], which gives a general architectural overview about *SeDiCo*, its central aspects and illustrates the relation to the structure of this chapter.

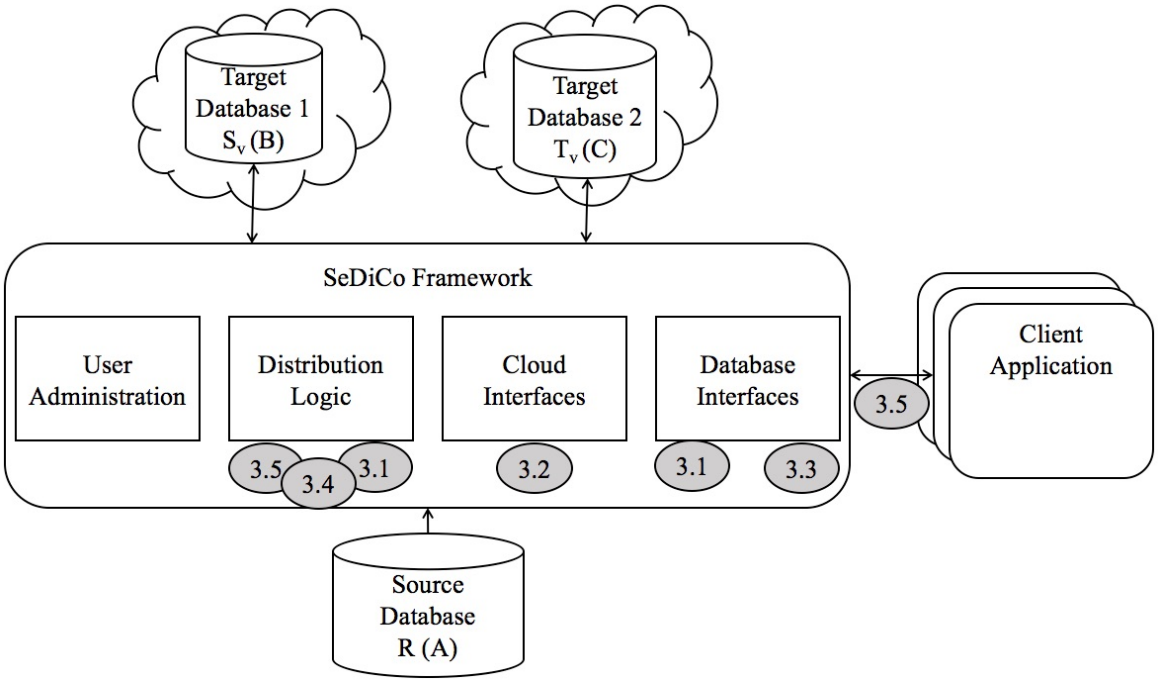



Figure 3.1: *SeDiCo* Architecture Mapped to Chapter Content

[1] and therefore implicitly for the query mechanisms developed in this thesis

[2] note that the user administration component is out of the scope of this work and is not described in more detail here. Section 3.1 is a central aspect in the distribution logic and in the database interfaces component and is therefore illustrated twice

First, security and privacy challenges are addressed in Section 3.1 with the *Security-by-Distribution* approach. This is motivated by the usage of Cloud Computing architectures[3] (cf. Section 3.2). The *Security-by-Distribution* principle with different database systems demands an abstraction layer that encapsulates different vendor-specific SQL implementations into a centralized interface. Therefore, object-relational mappers (ORMs) are in the focus of Section 3.3. Another key element is the investigation of the related work for the caching approach in 3.4. Last not least, Section 3.5 presents several alternative benchmarks with a strong focus on databases to evaluate the before-mentioned approaches.

## 3.1 Data Security and Privacy

With respect to current data security and privacy challenges that emerge with the usage of Cloud Computing, this section outlines basic definitions and relates them to the *SeDiCo* approach.

A basic definition of security is the preservation of confidentiality, integrity and availability of data (DIN ISO 27000, 2011). This is also known as the CIA-Principle (Confidentiality, Integrity, and Availability) based on the initial letters of the three protection criteria:

- Confidentiality: the preservation of confidentiality is to ensure, that data cannot be accessed by unauthorized entities.
- Integrity: according to the integrity feature, it has to be guaranteed that all data are correct and complete.
- Availability: is to assure that only authorized entities always have access and are always able to use data that they are authorized for.

Database security is commonly known as the preservation of the CIA-Principles in the database community and only systems that address all three dimensions can be called secure (Bertino & Sandhu, 2005) (Sion, 2007) (Mattsson, 2008) (Elmasri & Navathe, 2015). Further database-focused security challenges with a more detailed classification of threats and breaches (e.g. active, passive, direct and

[3] i.e. SaaS with Database as a Service (DBaaS), PaaS and with a specific focus on IaaS, as it offers the highest flexibility regarding the software stack, aiming at creating a framework for the secure and privacy-aware storage of data in especially public clouds. Furthermore, it is also applicable in private, community and hybrid clouds.

indirect attacks, etc.) that would go beyond the scope of this thesis are presented by (Rohilla & Mittal, 2013).

With respect to the focus of this work, it has to be stated that security relates to all kinds of data (i.e. general data) that is processed either by cloud consumers or by cloud providers. Thus, it is a broader definition than privacy, as privacy only considers personal data that are closely related to an individual or entity (Pearson, 2013). Above that, (Cloud Security Alliance, 2011) formulated a *Data Lifecycle* (Figure 3.2[4]) in order to maintain the CIA (Confidentiality, Integrity, and Availability) criteria with a concrete definition of each step. It further has to be noted that data can jump between the single phases without restriction and it is also possible that data not pass through all mentioned stages.

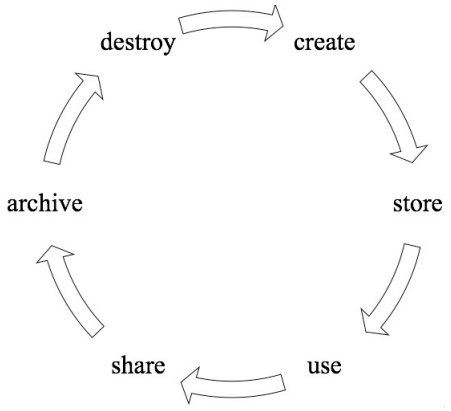



Figure 3.2: Data Lifecycle

### 3.1.1 Privacy

In contrast to security, the focus of privacy is on personal data that belong to or describe a concrete entity, i.e. a person or individual. Moreover, privacy is the control of data with respect to its collection, usage, disclosure and retention (Pearson, 2013). Closely aligned to this is the definition of the EU law in Directive 95/46/EC on the protection of personal data, which establishes that personal data mean any information related to an identified or identifiable natural person. Accordingly, an identifiable person is one who can be identified, directly or indirectly, in particular by reference to an identification number or to one or more factors specific to his physical, physiological, mental, economic, cultural or social identity.

[4]adapted from (Cloud Security Alliance, 2011)

Accordingly, (Pearson, 2013) states that there is a classification into sensitive data (e.g. credit card number, health data, financial data, etc.) and others (e.g. name, surname, address, etc.). On first sight, such a classification seems reasonable, however, this is considered highly context-dependent. There is no doubt that the previously mentioned examples of sensitive data are always sensitive data and in a telephone dictionary, the other data seems not that sensitive. However, for a thief that waits at the airport for a family that is going on holiday, information like name and address is critical and highly sensitive. Moreover, the OECD (The Organisation for Economic Co-operation and Development) Guidelines on the Protection of Privacy and Transborder Flows of Personal Data (OECD, 2013) summarized 8 principles (stemming from various different legal regulations of different countries worldwide) for the control and the management of personal data. As almost all developed countries have implemented these guidelines, they are shortly listed as follows:

1. Data Collection Limitation
2. Data Quality
3. Purpose Specification
4. Use Limitation
5. Security
6. Openness
7. Individual Participation
8. Accountability

Currently, concepts of data privacy refer to the right of an individual to be able to find out and to decide who will be able to collect, use and share which personal information. In the EU, these principles have been instantiated in the Data Protection Directive 95/46/EC (European Commission, 1995), which has been transposed into national law by all member states of the EU, and all EU members have harmonized data protection laws. Furthermore, this directive is currently under review and the current draft includes several novel aspects, including a *right to be forgotten, high standards for the creation of personalized user profiles, a*

*mandatory data protection impact assessment* and more effective means to impose sanctions on data protection violations.

Above that, there are anonymity and *pseudonymity* which refer to the *unlinkability* of private information with an individual or with an artificial identifier (e.g. a primary key). However, it is challenging to ensure anonymity for relational data while keeping its usefulness at the same time. Here, data schemes like column, document, key-value or graph stores provided by NoSQL architectures might help to implement an advanced level of *anonymity* and *pseudonymity*.

### 3.1.2 Implications for *SeDiCo*

With respect to the above-mentioned notions of security and privacy, *SeDiCo* ensures an improved level of security with its FVPD approach. Data cannot be used by unauthorized entities (confidentiality) due to the logical distribution of the partitions. Moreover, as FVPD data have to be *disjoint* and *complete* (cf. Section 1.1), their integrity is ensured. The availability is improved with the usage of different cloud infrastructures for the FVPD partitions. Regarding the privacy, it has to be noted that an improved degree of *anonymity* and *pseudonymity* is achieved through the FVPD approach. Although data still are identifiable through an artificial identifier (e.g. primary key), the distribution scheme of the partitions is unknown for an intruder and therefore it is not possible to combine the information that is contained in the FVPD relations. Such a distribution approach has already been proposed by (Aggarwal et al., 2005), but without going into further details on the architecture or the location of the different data storage locations (i.e. server or cloud infrastructures). Moreover, this work used a single database system, which does not allow to distribute data across different databases, which is a key feature of the presented *SeDiCo* implementation.

Additionally, *SeDiCo* with its FVPD approach offers the possibility to vertically partition data, such that the *anonymity* for e.g. queries on the public cloud partition is preserved (provided that the rows in this partition are only identifiable by their *artificial* primary key), while in combination with the other partition the *anonymity* is lost. This would allow 3rd parties to create *anonymized* statistics of the data without losing or lowering the level of security or privacy.

Yet, the level of privacy can be further improved with e.g. data encryption or with dynamic data schemes (NoSQL), however this is of minor interest in this work, as the response time should be improved here instead.

Table 3.1: Mapping the CIA-Principles to *SeDiCo*

| **Principle** | **Technique** | ***SeDiCo* Implementation** |
|---|---|---|
| Confidentiality | Encryption of data | Defined as future work task |
| | Encryption of communication channels | Encryption of communication channels (HTTPS, SSL, TLS, etc.) can be used. |
| | Authentication | Database authentication mechanisms can be used. |
| | Authorization | Database authorization mechanisms can be used. |
| | Partitioning | Vertical partitioning is used to preserve confidentiality and privacy through anonymization and pseudonymization. |
| Integrity | Authentication | Database authentication mechanisms can be used. |
| | Integrity Constraints | Defined as future work task |
| | Digital Signatures | Defined as future work task |
| Availability | Backup and Recovery | Database backup and recovery mechanisms with replication (in the cloud) can be used. |
| | Isolation Levels | Database isolation levels can be used. |
| | Proxy Servers | Proxy Servers can be used. |

Table 3.2: Mapping the Privacy Principles to *SeDiCo*

| **Principle** | **Technique** | ***SeDiCo* Implementation** |
|---|---|---|
| Privacy | Encryption | Defined as future work task |
| | Anonymization and Pseudonymization | Vertical partitioning is used to preserve confidentiality and privacy through the distribution of the partitions. However, unlinking the partitions from their identifying primary key (i.e. artificial identifier) is considered as a future work task |

Finally, it can be concluded that *SeDiCo* provides an advanced level of security and privacy features through its *Security-by-Distribution* principle. This advanced level comes from the separation of relational data into logical chunks that are (ideally) distributed across different cloud vendors and are thus *anonymized* and *pseudonymized* to a certain degree.

Another appealing feature of FVPD approach, is the *anonymization* and the *pseudonymization* of data (Bertino & Sandhu, 2005). Anonymization as well as pseudonymization in this context mean the elimination of all semantically and uniquely identifiable attributes that relate rows to a specific entity (e.g. a person). Yet, according to (Bertino & Sandhu, 2005) in nowadays web and cloud architectures, just erasing entity-identifying attributes and storing them at different locations connected via their primary keys, is not sufficient, as there are data mining technologies (e.g. statistical analyses) (Sweeney, 2002) (Sion, 2007) that enables intruders to deduce further sensitive information. This is perfectly understandable in scenarios where parts of the data are openly accessible for everybody[5], as data mining approaches rely on such additional open data. This should of course be taken into concern before the actual partitioning in *SeDiCo* (if open accessible data are involved[6]).

## 3.2 Cloud Computing

Considering huge amounts of data and the need for storing and analyzing them, renting computing and storage resources from external providers is an auspicious way to minimize costs. Moreover, for Cloud Computing vendors, the pooled usage of virtualized resources that abstract from the physical hardware layer promises a permanent utilization and a stable cash flow. This is basically the idea behind Cloud Computing, as the following National Institute for Standardization and Technology (NIST) definition shows.

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* (Mell & Grance, 2011)

Moreover, NIST defined a reference architecture that outlines the key actors in Cloud Computing as consumer, provider, broker, auditor and carrier. These actors and their connection with each other are further illustrated in Figure 3.3[7].

[5] e.g. in anonymized surveys or statistics

[6] An application scenario where this is actually an issue can be found in the author's previously published work (Kohler, Simov, Fiech, & Specht, 2015)

[7] adapted from (Bohn et al., 2011)

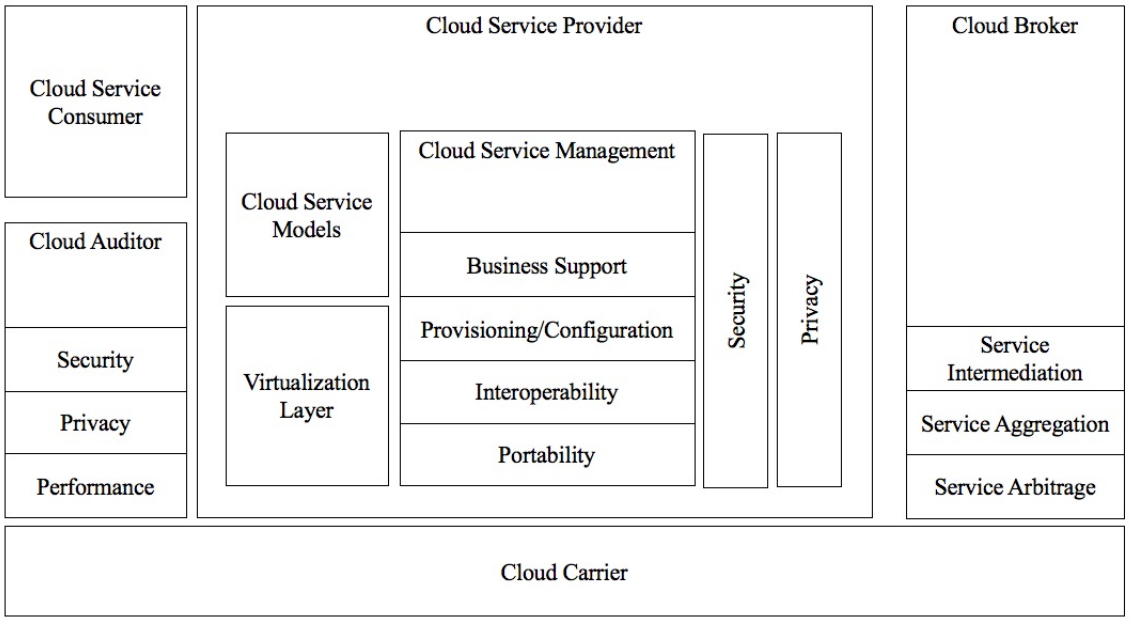



Figure 3.3: Cloud Reference Architecture

Going from left to right, there is the *Cloud Service Consumer* (short consumer) that buys and uses a service provided by the *Cloud Service Provider* (short provider) or by a *Cloud Broker* (short broker) if the service integration is too complex for the consumer. A *Cloud Auditor* (short auditor) assesses services based on their *Service-Level-Agreements* (SLAs), which include the monitoring and the documentation of the service usage. The provider offers services in *Cloud Deployment Models* and realizes services on virtualized infrastructure resources, subsumed as *Virtualization Layer* in Figure 3.3. Moreover, the provider is able to offer support for the services, and is able to provide service configuration opportunities (depending on the service models) and interoperability and portability interfaces. This is summed up as *Cloud Service Management*. Located at provider-side, there are security and privacy issues (cf. Section 3.1, e.g. Identity Management, Access Management, Encryption, etc.), as the provider is responsible for them. On the right side, a broker can be used by consumers to integrate or compose complex services. If a broker is used, it acts like a proxy between consumer and provider, which eases the service management for the customer. In the end, a *Cloud Carrier* (short carrier) acts as a medium of transport for service requests and the corresponding answers between the consumer and the provider.

Closely related to this reference architecture, NIST identified 5 key characteristics of capabilities that are offered as cloud resources, cloud services or cloud capabilities:

- *On-demand self-service*, which enables consumers to use cloud capabilities fully automatically without any human contact on the provider-side.
- *Broad network access*, which requests that a consumer has a proper network bandwidth to and from the cloud.
- *Resource pooling*, which defines that all cloud capabilities are virtualized and dynamically allocated from physical (cloud provider) resources on demand (i.e. multi-tenancy).
- *Rapid elasticity*, which enables dynamic scalability of the rented cloud capabilities for the customers.
- *Measured service*, which determines that the capabilities are automatically monitored and checked (for the consumers and the providers) according to their Service-Level-Agreements (SLAs).

### 3.2.1 Service Models

NISTs reference architecture contains 3 *Service Models*, in which a cloud service with the 5 above-mentioned characteristics could be offered: *IaaS*, *PaaS* and *SaaS*. These are now described in more detail.

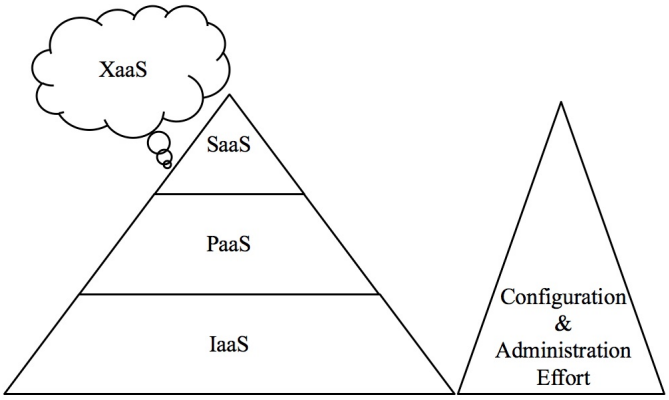



Figure 3.4: Cloud Computing Service Models

Figure 3.4[8] shows that in the basic layer IaaS, the cloud consumer has direct access to storage, network and the operating system of the virtualized resources, but not to the underlying physical hardware. As the upper layers depend on this IaaS layer, the cloud completely abstracts from the underlying physical capabilities with the usage of virtualization. One layer above, on PaaS the consumer is able

[8]according to (Mell & Grance, 2011)

to directly deploying his applications to a cloud platform. On the SaaS layer, the consumer is only able to access already deployed applications, mostly provided as services. This also explains the configuration and administration effort required for each Service Model from high (IaaS) to low (SaaS). The SaaS layer perfectly sketches the idea of the so-called Everything as a Service (XaaS) model (Hofmann et al., 2009). The idea behind XaaS is to provide all cloud resources as services, according to a service-oriented architecture (SOA), to be able to measure and compare similar services to each other, towards a standardization of cloud services.

### 3.2.2 Deployment Models

NIST's definition also includes the differentiation between *public*, *private*, *community* and *hybrid* clouds as so-called *Deployment Models*.

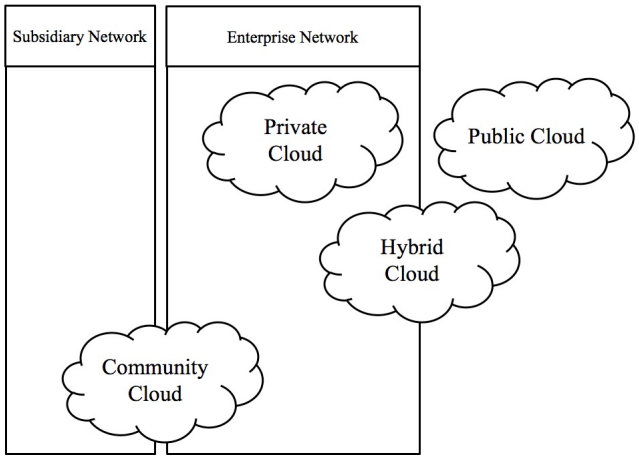



Figure 3.5: Cloud Computing Deployment Models

According to Figure 3.5[9], a private cloud is hosted, provided and operated inside the internal network, whereas a public cloud does not belong to the internal network and is managed by a third party. Consequently, a hybrid cloud is a mixture of both, public and private cloud with an intermediate layer in between. This intermediate layer[10] connects the two physically separated clouds for data and application integration purposes. Lastly, there is the community cloud, where only a closed group has access to. Such a cloud might be hosted by one participating community, but also by an external third party (illustrated as the gap between the enterprise network and the subsidiary network in Figure 3.5) (Mell & Grance, 2011).

[9] according to (Mell & Grance, 2011)

[10] also known as abstraction layer, that encapsulates different APIs into one unique interface

### 3.2.3 Implications for *SeDiCo*

*SeDiCo* enables the usage of different clouds in an *on-demand self-service* manner. The user of the framework just defines the FVPD partitioning schema, the clouds that should be used for the storage of the data and the respective user credentials for the clouds. All other activities, i.e. the partitioning, the data transfer, the instantiating of the (virtualized) cloud resources, etc. are automatically handled by the framework. Depending on the concrete amount of data that should be partitioned and transferred to the clouds, a *broad network access* as defined as the second criterion above, is required. Here, the *SeDiCo* has to rely on the existing network infrastructure and bandwidth. The framework relies on *resource pooling* techniques offered by the respective cloud providers. This depends on the used cloud *Service Model* (see below) and whether ready-to-use database instances (e.g. in a SaaS model) or just hardware (e.g. in an IaaS model) is utilized. This also holds for the *rapid elasticity*, as it depends on how fast the cloud provider is able to scale the rented capabilities. Above that, the monitoring of the used cloud capabilities (i.e. the *measured service* criterion) is dependent on the possibilities and tools that are offered by the provider.

The above-mentioned deployment models clearly illustrate the major drawbacks of especially public clouds: the unsolved data security and privacy aspects while using foreign cloud capabilities and the challenging task of integrating different cloud interfaces, in order to be independent from respective cloud vendors (i.e. vendor lock-in[11]).

These challenges (and how they are addressed in *SeDiCo*) can be subsumed under the following bullet points:

- Vendor Lock-in

  The *SeDiCo* framework clearly shows how different cloud deployment models are usable while addressing cloud security and privacy concerns and the dreaded vendor lock-in. In this context vendor lock-in means that the effort to change a certain cloud provider is bigger or more expensive in terms of money than to bear its disadvantages (e.g. higher prices, slower performance, poor service, etc.) (Binz et al., 2012). (Arora & Gupta, 2012) also define the vendor lock-in as a major challenge where consumers are enabled to change cloud vendors easily with little effort.

[11] also known as *provider lock-in*

- Cloud Abstraction

  In order to minimize the effort of developing and encapsulating different cloud interfaces, *SeDiCo* relies on *jclouds* (Apache, 2016b) as a cloud abstraction framework. (Kaiser, 2013) showed that jclouds is the most suitable framework (among others like *libCloud* (Apache, 2016e) or *deltaCloud* (Apache, 2016d)) to handle different cloud APIs. Based on the above-mentioned cloud abstraction work, it is possible to integrate various different cloud providers into *SeDiCo* with little implementation effort. The reason for jclouds was the support of various cloud vendor APIs (an overview can be found in (Apache, 2016b)) and the corresponding Java integration capabilities.

- Data Security and Privacy in the Cloud

  Generally, state-of-the-art security deals with encryption of data and its corresponding key management (Steve & Ushar, 2011). Beyond providing secure access to virtual cloud resources, they aim to encrypt concrete distributed database data, which is transferred between customers and the cloud. Concerning data distribution approaches over various clouds including data distribution, replication and encryption, a well-documented approach was developed by (Neves et al., 2013). In contrast to this, *SeDiCo* uses vertical data partitioning and it supports different database systems. Hence, it is considered more flexible concerning the underlying IaaS infrastructure and therefore easier to extend.

## 3.3 Object-Relational Mapping (ORM)

Object-relational mapping (ORM) is typically used to overcome the gap between the relational and the object-oriented paradigm, and this gap is of particular interest in *SeDiCo*, as it is implemented in Java, as an object-oriented programming language. In the relational model (Codd, 1970), there are rows with attributes stored in relations and different relations are connected with each other through referential integrity constraints, i.e. foreign keys. This is also known as the persistence layer in software development (Bauer et al., 2007). On the contrary, there are objects that represent real world behavior. These objects also have attributes but are created (instantiated) from classes that represent real world items (or things). Classes are also connected via relationships, i.e. generalization, inheritance and associations (aggregations or compositions). Although there are

similarities between rows, relations and objects and their classes (i.e. attributes and relationships), their main difference is the fact that rows represent data (*a row is a statement of truth*) (Cattell, 1994) that builds the foundation of each application and objects represent behavior (Ottinger et al., 2015). With respect to this, the differences between the two paradigms can be summarized as the well-known *Impedance Mismatch*, extensively described by e.g. (Cattell, 1994). Figure 3.6[12] illustrates this in a simplified architectural overview, where the ORM layer is located between an application and several databases.

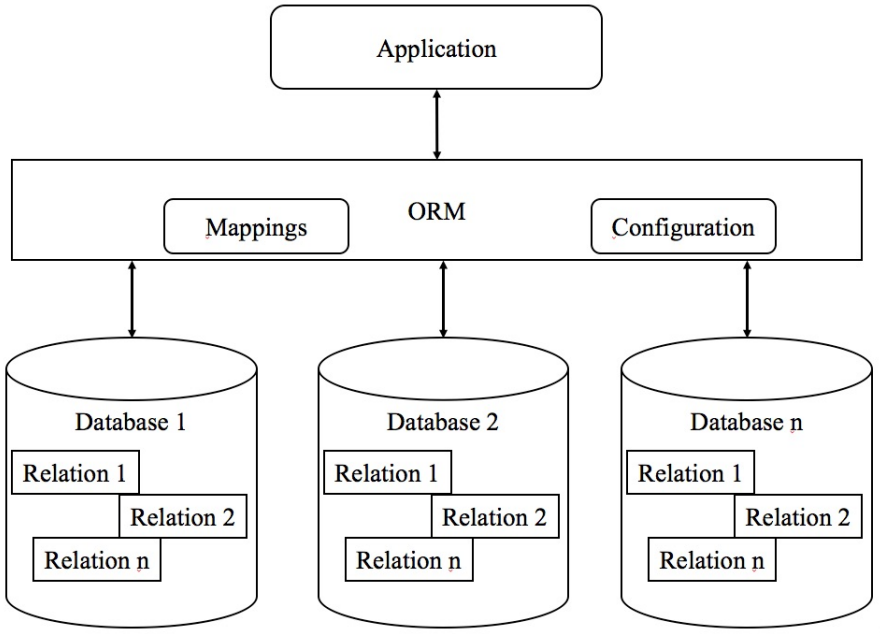



Figure 3.6: ORM Architecture

### 3.3.1 Impedance Mismatch

(Ireland et al., 2009) categorized and summarized the main challenges of the impedance mismatch shortly but vividly and proposed a framework to address the main issues caused by the paradigm mismatch. Their main contribution can be summarized as follows (Table 3.3[13]):

Table 3.3 shows that in order to overcome the impedance mismatch, ORM frameworks use mappings (in form of e.g. XML files or programming annotations, etc.) between classes, relations, rows, and objects. Thus, meta data that describe relations are used to map relational to object-oriented concepts.

[12]adapted from (Van Zyl et al., 2006)
[13]adapted from (Ireland et al., 2009)

Table 3.3: Impedance Mismatch Challenges

| No. | Challenge | Object-oriented paradigm | Relational paradigm | ORM approaches |
|---|---|---|---|---|
| 1 | How to build a class hierarchy? | Class hierarchy (generalization, inheritance, multiple inheritance) | As there is no hierarchy concept or model to build a hierarchy of relations, it is not possible | Multiple approaches: mapping by single table, table per class, one to many, one to one, many to many mappings, etc. (Russell, 2008) |
| 2 | How to map the dynamic behavior of objects to static rows? | An object represents behavior by its methods and thus the object might change its state during its lifetime | *A row is a statement of truth* (Cattell, 1994), its values can be changed but a row does not have any methods that belongs to it | Mapping of object attributes to rows and use object methods to change the corresponding rows |
| 3 | How to ensure consistency between objects or between rows? | Information Hiding (hide object attributes behind getter and setter methods) | Referential Integrity, i.e. primary and foreign keys, unique constraints, etc. | Mapping of referential integrity constraints to classes, i.e. one to many, one to one, etc. |
| 4 | How to ensure identity between objects and rows? | Objects are uniquely identifiably by their object id (OID), however the OID is independent of the single object it identifies | Rows are uniquely identified by their primary key, but primary key must be part of the row it identifies | Mapping of the primary key to the corresponding object attribute to shift the uniqueness from primary key to OID or another dedicated object attribute |
| 5 | How to realize different ways of accessing rows and objects? | Objects are instantiated based on classes and accessed by their methods | There is a set-based access based on SQL as a query and manipulation language, which returns rows in sets | Mapping of relational sets to object-oriented sets, (i.e. lists, result sets, etc.) |
| 6 | How to manage different types of modeling data? | A class model (e.g. UML) is designed specifically for a single application | A data model (e.g. ERM) might be the foundation for various applications | There are multiple approaches in the current state-of-the art: map one data model to n class models ($1 : n$) for the sake of reusability or other approaches such as (Melnik et al., 2008) create an $n : m$ relationship between data and class model (implemented as database views), which enables and facilitates maintenance of both models |

Another appealing advantage of this mapping is that it abstracts from a concrete database or SQL implementation. Hence, it hides all relational concepts and the corresponding SQL and uses object-oriented concepts instead. Because of this abstraction from concrete databases and their specific SQL implementations, various different databases can be used without changing the application logic (even simultaneously in *SeDiCo*).

Besides other advantages of ORMs like rapid prototyping, better maintainability of application logics through convention over configuration, there are disadvantages like a high learning curve and more important, the previously described database abstraction (via mappings) causes a big overhead for the communication between the application logic, the ORM and the databases. This results in slower performance when an ORM is used (Ottinger et al., 2015).

In summary, it can be noted that using an ORM is twofold: on the one hand, the database abstraction is the crucial feature and advantage for its usage, but on the other hand using an ORM means performance degrades for accessing and manipulating data. However, the *SeDiCo* framework heavily depends on database abstraction, as it supports different databases and their specific SQL dialects. Based on this feature, relations can be partitioned vertically and the partitions can be stored in different databases. In addition to this, cloud abstraction (cf. Section 3.2.3) follows the same principle. It encapsulates different cloud application programming interfaces (APIs) into one unified layer.

Conversely, these database and cloud abstraction advantages suffer from tremendous performance losses. Firstly, the ORM overhead is responsible, but secondly, the fact that different databases can be used simultaneously for the vertical partitions in *SeDiCo* is another issue that decreases the response time. Thirdly, there is the network overhead to and from the different clouds. (Van Zyl et al., 2009) confirmed this when they regarded the ORM as an additional intermediate layer between databases and applications and therefore, ORMs are very likely to become a bottleneck in a setup described in Figure 3.6.

After this generic introduction of ORMs, the following section now outlines Hibernate as a concrete ORM implementation that is used in *SeDiCo* and gives a more detailed view on the performance issues (e.g. n+1 selects problem, caching, etc.) caused by the abstraction mechanisms. The section starts with a short introduction of Hibernate and discusses some implications, regarding caching and the *Hibernate Query Language (HQL)*, etc. that have emerged during its usage as well as possible alternatives.

### 3.3.2 Hibernate as an ORM Implementation

During the last few years various ORM implementations (Linq (Microsoft, 2016), Hibernate (RedHat, 2016), SQL Alchemy (SQLAlchemy, 2016), etc.) for the

Listing 3.1: HQL n+1 Selects Problem

```
1 List<Customers> customers = session.createCriteria(
2         Customer.class).list();
3         for (Customer c : customers) {
4                 c.getOrders();
5         }
6 }
```

Listing 3.2: Customer Select Query

```
1 SELECT * FROM CUSTOMER;
```

plethora of programming languages have been developed. Due to its widespread usage in industrial and scientific contexts and due to its advanced development, Hibernate was chosen as an ORM implementation in *SeDiCo*. Hibernate was founded in 2002 by Gavin King (Bauer et al., 2007) and is currently the most used framework (Ottinger et al., 2015).

Above that, Hibernate is used for the evaluation of the implemented query mechanisms in Section 6. Therefore, a more detailed description of the framework is given here.

### N+1 Selects Problem

This challenging issue comes from the impedance mismatch and refers to the challenge of access methods (cf. Table 3.3). The following example illustrates the problem.

Listing 3.1 shows an HQL query that searches for all *CUSTOMERs* and their *ORDERS* in two relations. Now, the Hibernate framework transfers this HQL query into the SQL query in Listing 3.2.

As soon as all *CUSTOMER* rows are collected in the *CUSTOMERS* list (1), there is a call for every *CUSTOMER* (2) to retrieve its *ORDERS* (3).

Accordingly, in (line 3, Listing 3.1) for each *CUSTOMER* a

Listing 3.3: Order Select Query with Criteria

```
1 SELECT * FROM ORDERS WHERE customerID = ?
```

Listing 3.4: Customers and Orders Join Query

```
1 SELECT * FROM CUSTOMERS JOIN ORDERS ON customers.customerID =
    orders.customerID;
```

query is issued against the database. This results in n select statements for all orders and in 1 select statement to collect all *CUSTOMERS*, which are $n + 1$ selects. In contrast to this, in the relational model, this scenario would have been realized with a single join query, e.g.

and this results in faster response times.

### 3.3.3 Implications for *SeDiCo*

Comparing Listing 3.1 and Listing 3.4 it can be concluded that the SQL in Listing 3.4 is more effective than the Java representation depicted in Listing 3.4. However, the SQL representation is bound to a specific database implementation[14], whereas the Hibernate's Java representation is independent from a specific database implementation. Thus, different database implementations become exchangeable, or as it is the case in *SeDiCo* are simultaneously usable for different partitions. This, besides the widespread usage of Hibernate are the main reasons why *SeDiCo* uses Hibernate as the ORM framework.

## 3.4 Caching

This section introduces caching as an approach to improve the response time of FVPD data with respect to hypothesis 2 and the *caching* approach. After a short overview about cache memory hierarchies and their performance, a general middle-tier caching architecture is outlined in Section 3.4.1. Then Section 3.4.2 discusses two basic requirements for a caching architecture and the following section discusses a concrete cache workflow, followed by cache implementations. After that, different caching schemes are outlined in Section 3.4.4 which also focuses on the challenging task which data should be cached and which not. Finally, Section 3.4.5 outlines state-of-the-art caching approaches with a strong focus

[14] although SQL is a standard, there are different database-dependent implementations (e.g. MySQL, Oracle, Microsoft, etc.)

on data selection and on cache coherence protocols (i.e. cache synchronization, replication and invalidation).

The presented caching approach in this work is similar to the principle of CPU caches, where a little amount of memory located besides the CPU increases the CPU performance with preventing it from constantly accessing the slower (secondary or tertiary) memory to execute its instructions. (Garcia-Molina et al., 2008) provide a good overview about the storage components of a system, e.g. a database server. Figure 3.7[15] enhances this overview with corresponding memory access times.

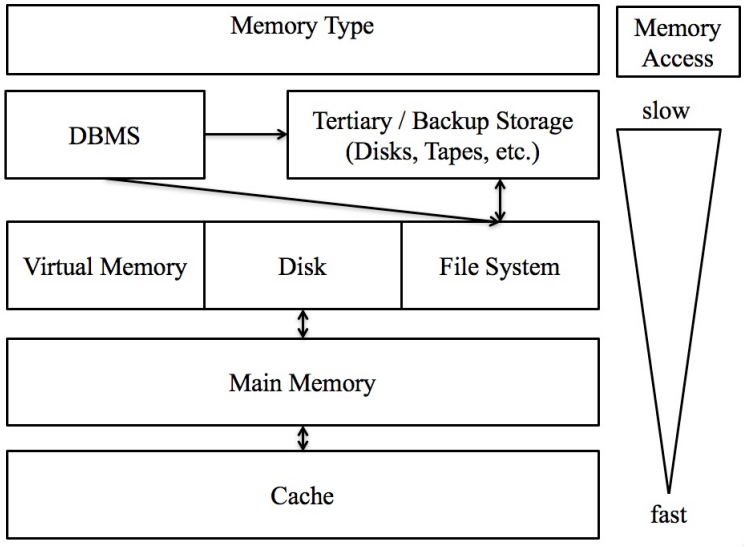



Figure 3.7: Cache Hierarchy

If a CPU finds the requested information in the cache, it reads it directly from there without accessing slower memory types. This is known as a cache hit, whereas a cache miss occurs, if the CPU cannot find the information in the cache. Cache memory is fast but expensive (in terms of money) compared to other memory types. Yet, it only has a small storage capacity and if the cache is full, replacement strategies such as *first-in/first-out*, *least recently used* or *least frequently used* (Davision, 2001) have to be implemented. (Franklin et al., 1997) define a cache as a *"dynamic form of data replication"* (Franklin et al., 1997).

In addition to this, replication is the duplication of data to e.g. different locations in order to prevent data access failures or server outages (Garcia-Molina et al., 2008). Multiple copies of the data result in several challenges concerning the consistency of data (ACID criteria), their distribution and their synchronization.

[15]adapted from (Garcia-Molina et al., 2008)

The focus of this work is on so-called *middle-tier database* or *application-level* caches, as they have proven to solve database bottlenecks in large distributed database architectures.

### 3.4.1 Middle-tier Database Caching

*Middle-tier database caches* are placed between the application and database layer and that is why they are able to improve the performance of both layers (Ports et al., 2010). This advantage was also recognized by (Bornhövd et al., 2004), as caching on database layer allows all upper layers to benefit from its advantages (e.g. performance, etc.). However, depending on the synchronization strategy they do not guarantee all ACID criteria, which possibly leads to inconsistencies in the database (Ports et al., 2010).

The basic idea is to place the cache in the main memory either on a dedicated server or as a client-based cache between the databases and the clients. There are several ways of implementing a cache, i.e. as a *forward proxy* between the clients and the application logic, as *reverse proxy* between the databases and the application logic or as *interception proxy* directly into the application layer (Hofmann & Beaumont, 2005). This is further illustrated in Figure 3.8.

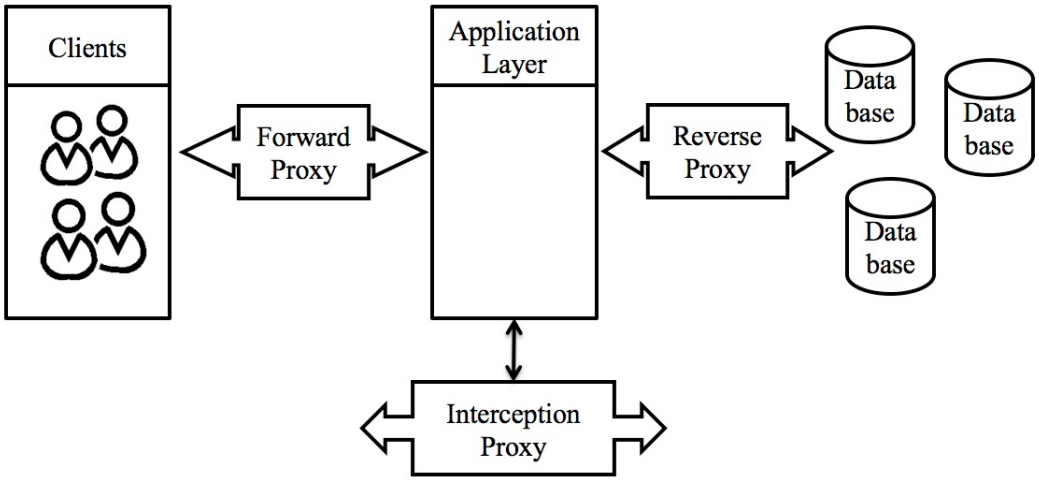



Figure 3.8: Cache Positions

Analogous to the above-mentioned CPU cache mechanisms, if a query can be served from the cache, there is a fast *cache hit* in contrast to a slow *cache miss* that demands loading the requested data directly from the FVPD partitions.

### 3.4.2 Requirements for a Cache Implementation

The introduction of a cache is based on the following two requirements, according to (Luo et al., 2002):

1. Both, the database schema and the application layer remain unchanged
2. The cache must provide a sound and realistic data manipulation performance, compared to a database schema that is not partitioned and operated without a cache

Considering the first aspect, the integration of the proposed middle-tier database caches (called proxies, cf. Figure 3.8) requires changes in the application layer (i.e. the *SeDiCo* client). However, with respect to the aimed performance improvement, changes are comprehensible, hence the other requirement can be fulfilled. More requirements as mentioned in (Luo et al., 2002) like high availability, failover, etc. are not in the scope of this work and therefore not discussed any further.

### 3.4.3 Cache Workflow

Figure 3.9 gives a rough sketch of a caching workflow including 4 basic steps that are involved with the usage of a cache.

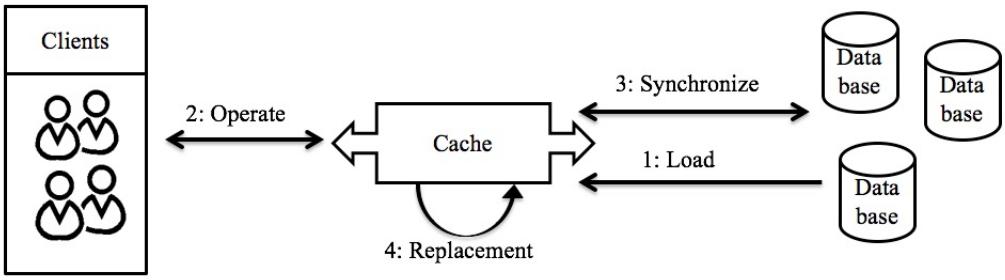



Figure 3.9: Cache Workflow

Figure 3.9 shows that if data are loaded into the cache (step 1), clients are able to directly access them (step 2). This is then a *cache hit*, whereas data that are not in the cache produce a *cache miss*. With respect to data manipulations, data between the cache and the databases have to be synchronized either time-interval based, volume-based or user-triggered, etc (step 3). If the cache is full (in terms of its storage capacity), previously cached data have to be removed (step 4).

### 3.4.4 Caching Schemes

The analysis of a caching mechanism includes 4 different caching techniques, known as *caching schemes*[16] (Luo et al., 2002). This includes the caching of:

1. an entire table, called *table caching*
2. only a part of the table data (e.g. in a view), called *subset table caching*
3. an entire query result, called *query caching*
4. an intermediary query, called *intermediate query caching*

Furthermore, (Luo et al., 2002) discuss table caching with respect to cache hits, cache misses, maintenance, replacement strategies, etc. However, they only consider table caching in their work. In order to determine an adequate caching scheme in the context of this thesis, the above-mentioned schemes have to be investigated in closer detail.

A deeper analysis of these caching schemes shows that *query caching* and *intermediate query caching* can be subsumed under the same approach. Here, either an entire result set (3) or an intermediate result set (4) is cached. However, caching query results is heavily dependent on the respective database workload.

On the contrary, there are approaches that load entire tables (1) or subsets of them (2) into the cache. Especially with small cache memories, *subset table caching* becomes a feasible approach. On the one hand, the probability of *cache misses* grows the smaller the cache memory is, but on the other hand, loading entire tables into the cache (*cache warming*) or updating the cache memory becomes faster.

To sum up the caching schemes, it can be concluded that

- the focus of this work is to evaluate caching approaches for FVPD data and to develop a basic performance metric which serves as a guideline for a great variety of different application scenarios. This performance metric should not contain any side-effects such as network overhead or overhead concerning the cache coherence protocols. Also, the aim is to achieve an easily comprehensible performance metric whose results should be easily reproducible.

[16] here, a *table* relates to a *relation* according to Codd's relational model, cf. Table 1.1

- *subset table caching*, *query caching*, and *intermediate query caching* would require adequate data selection strategies that define which data should be cached and which not. As such strategies are heavily dependent on the respective use case and application domain, such approaches would restrict the generalization of the evaluation results.

To sum up this caching section so far, Table 3.4 concludes all mentioned approaches and relates them to a possible application in the context of *SeDiCo*.

Table 3.4: Applicable Caching Approaches for *SeDiCo*

| **Caching Approach** | **Suitability for *SeDiCo*** |
|---|---|
| Memory Type | Main Memory, because of the current development between storage capacity and price |
| Middle-tier database cache | Implemented as a *forwared proxy*. As the cache resides in the main memory, differenced between *forward*, *intermediate*, and *reverse proxies* are considered minimal and therefore out of the work's scope. |
| Requirements for a cache implementation | First requirement (database scheme) remains unchanged can be fulfilled, due to the *forward proxy* implementation, changes in the application logic are minimal and the data manipulation performance is out of this work's scope. |
| Cache Workflow | *SeDiCo* follows the workflow depicted in Section 3.4.3: 1. cache warming, 2. clients operate on cache (cache hit/miss), 3. synchronization, and 4. replacement are out of this work's scope |
| Caching Scheme | Table caching promises best performance gains according to analysis in Section 3.4.4 |

### 3.4.5 Implications for *SeDiCo*

Further current state-of-the-art challenges, important not only in the context of *SeDiCo* but in all scenarios, where a cache is involved (e.g. client and server-based caching of web sites or caching frequently used applications in the RAM, etc.) are now briefly outlined. These challenges are of particular importance for the future development of the *SeDiCo* framework and include data selection approaches that define which data should be cached and which not, cache replacement strategies that control which data should be evicted in case the cache memory is full and cache synchronization mechanisms that determine when the cache memory should be updated.

### Data Selection for Caching

Closely related to the 4 above-mentioned caching schemes the challenging question which data to cache and which not is a current state-of-the-art research problem. This challenge originates from small and expensive main memory storages and is still relevant to firstly minimizing the required main memory and secondly, to improving the performance of In-Memory databases.

Caching suitable data, that is potentially used in the near future, depends on the respective database workload (Podlipnig & Böszörmenyi, 2003), but no one (to the best of the authors knowledge) has evaluated the caching architectures mentioned in this work with respect to the FVPD approach. This challenge is also addressed with the distinction between hot and cold data (Plattner, 2013). According to this, hot data are current data that are actually used and manipulated. Therefore, hot data have OLTP character. On the other hand, cold data are data that have already been processed and that are not very likely to be manipulated or changed anymore.

A similar but more sophisticated challenge with respect to data security and privacy, is the distinction between critical data that under no circumstances are allowed to leave the enterprise network (i.e. private data for a private cloud) and less-critical data that can be stored in a public cloud. As this challenge does not directly affect the caching focus of this work, it is considered as a future work task.

### Cache Replacement

(Podlipnig & Böszörmenyi, 2003) present an exhaustive overview about commonly used cache replacement strategies that are necessary if a cache is full and objects have to be replaced. In recent years, only little attention was paid to cache replacement strategies. The major reasons (among others) were decreasing storage costs and increasing storage volumes (Podlipnig & Böszörmenyi, 2003). Nevertheless, with the upcoming Big Data challenges, these problems become crucial again. For the sake of brevity, not all strategies, but the most general ones are listed and explained in Table 3.5.

The investigations in (Podlipnig & Böszörmenyi, 2003) demonstrate that there is no optimal strategy, that outperforms all others. Moreover, the strategies

Table 3.5: Classification of Cache Replacement Strategies

| Caching Classification | Cache Replacement Strategy | Description |
|---|---|---|
| Recency | LRU | The least recently used row is removed. |
| | CLOCK | Rows are stored as a ring list with a LRU bit (R bit), if the R bit is 1, the row was recently used. Rows with an R bit of 0 are removed from the cache. A more detailed description can be found in (Fan et al., 2013) |
| Frequency | LFU | The least frequently used row is removed. |
| | Perfect LFU | Requests to a row are counted, even if the row is removed, the counter remains. |
| | In-Cache LFU | Requests to a row are counted, if the row is removed, the counter is also removed. |
| Size | SIZE | The row that needs the largest storage space in the cache is removed. |
| Costs | CERA (Ayani et al., 2002) | Costs in terms of effort (e.g. response time) for accessing a row in the cache are compared to costs for accessing a row in the original database. |
| Modification Time | MRU | Most recently used (modified - in contrast to frequency-based algorithms) rows are removed from the cache. |
| Expiration Time | TTL | A timer (time to live) indicates when a row should be removed from the cache. |
| Random Values | RAND | Rows are removed randomly from the cache. |

heavily depend on the underlying database volume and the respective workload of the clients.

### Cache Consistency Models

Basically, there are *ACID* and *BASE* also known as *strong* respectively *weak* consistency models. It has to be noted that the focus of the two models differ: firstly, there is ACID which refers to relational databases where data consistency is a key requirement. Secondly, there is BASE which was introduced with the upcoming NoSQL database architectures. This model refers to a weaker consistency, as it is based on Brewer's CAP-Theorem (consistency, availability, partition tolerance) which proved that in a distributed system only 2 out of the 3 CAP properties can

be fulfilled at the same time (Gilbert & Lynch, 2002). Table 3.6 distinguishes these 2 consistency models and illustrates their basic differences.

Table 3.6: Cache Consistency Models: ACID and BASE

| **ACID** | **BASE** |
|---|---|
| **Hard consistency** | **Weak consistency** |
| *A*tomicity: a transaction (considered as a basic set of database operations) is either performed entirely (commit) or not at all (rollback) | *B*asically *a*vailable: there will be a response to any request but data might be inconsistent (at cost of consistency) or the request might be delayed e.g. with an error message (at cost of availability) |
| *C*onsistency: the database system is in a consistent state before and after each transaction | *S*oft state: the state of the system might change over time even without any user input due to the eventual consistency property |
| *I*solation: transactions are independent from each other and cannot access data that is simultaneously processed by another one | *E*ventual consistency: guarantees that the system becomes consistent over time, i.e. the data is sooner or later propagated to all participating database nodes |
| *D*urability: data is long-lastingly stored in a database | |

This consideration shows that the 2 models mutually exclude each other and a decision for a consistency model has to be made in advance. Especially in data caching, the decision for a concrete consistency model particularly depends on the respective use case and on database workload. Hence, no clear recommendation for a consistency model can be given here. However, as the BASE model is focused on distributed systems, it can be stated that architectures based on the consistency model are extendable[17] more easily and this extensibility feature is a key factor for the performance gains when data are accessed and also for the management of bigger data volumes.

All in all, it can be noted that with respect to a cache implementation, both consistency models are considered feasible and therefore, concrete implementations, based on *SeDiCo*'s *Security-by-Distribution* approach are necessary to get comparable performance metrics.

[17] e.g. with horizontal scaling in which more computing nodes are added to the distributed system and data are replicated to the nodes automatically

### Cache Synchronization Strategies

In literature, caching and replication are often used synonymously (e.g. (Olston & Widom, 2002) (Sivasubramanian et al., 2007) (Garrod et al., 2008)). Basically, data in a cache are considered as *"dynamic replicas of the original data"* (Franklin et al., 1997). In the same manner, caching and replication are used synonymously in the remainder of this work. The main goal of caching is to improve the response time and with respect to replication, to increase a systems availability and scalability. Essentially, there are two fundamental approaches, how to preserve consistency across these multiple copies of the data: eager update propagation that ensures all ACID criteria and lazy update propagation that refers to the BASE model with its weaker eventual consistency (Pritchett, 2008) (Özsu & Valduriez, 2011) (Pritchett, 2008) (Özsu & Valduriez, 2011) (Garrod et al., 2008).

Especially in the context of mobile systems, data or cache synchronization becomes a crucial aspect, as a constant network connection cannot be guaranteed; here, there are two approaches to keep several distributed data sets up-to-date: conservative (based on eager) and optimistic (based on lazy) synchronization concepts. However, due to the uncertainty of mobile network connections, conservative strategies are not suitable for mobile clients (Lutteroth & Weber, 2009).

Another interesting concept, based on *master-slave replication* comes from version control systems such as GitHub (GitHub, 2016). Here every client has its own copy of the data in a local repository. Only data from this local repository are manipulated and later on committed to the remote (master) repository that maintains a history of all changes. However, if two clients manipulate the same data, their changes have to be merged together (i.e. synchronized) manually. Accordingly, (Lutteroth & Weber, 2009) proposed a concept (PDStore), where data is synchronized from time to time (i.e. incrementally) based on additional unique identifiers (GUIDs) for rows.

### Cache Coherence Protocols

In order to propagate data manipulations to the original database or to other caches, generally two propagation concepts can be used. On the one hand, there is a *centralized* (Figure 3.10, *master-slave replication*) and on the other hand a *decentralized* (Figure 3.11, *decentralized replication*) approach.

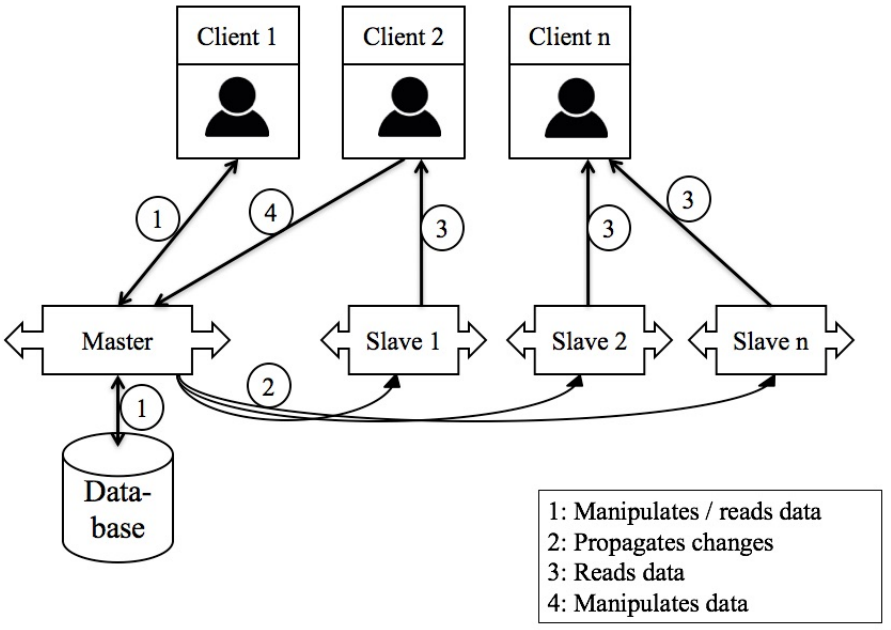



Figure 3.10: Mater Slave Replication

In a *centralized* or *master-slave* environment, there is a master that receives all manipulations. It then propagates the manipulation to the other caches (i.e. slaves). The slaves cannot receive manipulations, clients can only read data from them. This results in the following advantages and disadvantages[18]:

Table 3.7: Master-Slave Replication Discussion

| **Pros** | **Cons** |
|---|---|
| The propagation logic of data manipulations is easy, as only the master receives manipulations and forwards them to the slaves. | Master is likely to become the bottleneck because of the manipulation and synchronization effort. |
| The slaves do not need any synchronization logic as the master propagates all changes. | The slaves can contain outdated data, if the master has not propagated changes yet. |
| It is guaranteed, that at least the master contains current data. | Master is a single point of failure. |

In contrast to the above-mentioned *master-slave replication*, in a decentralized replication environment, all caches are allowed to receive and propagate data manipulations. As illustrated in Figure 3.11, data manipulations are propagated to the other caches or to the original database from that cache, where the data were manipulated.

A more detailed analysis of corresponding replication protocols (e.g. Single Master with Limited Replication Transparency or Primary Copy with Full Replication Transparency, etc.) that implement the above-mentioned concepts, can be

[18]adapted from (Özsu & Valduriez, 2011)

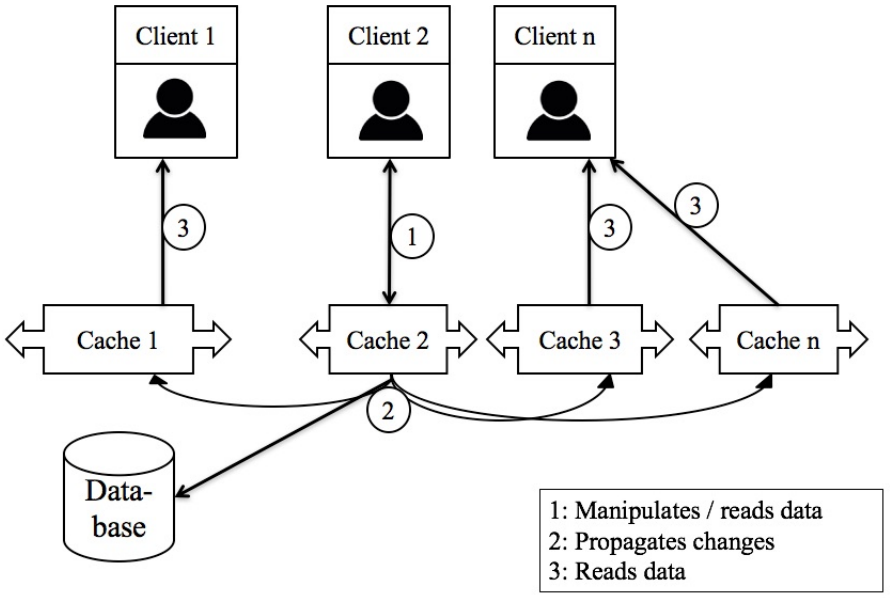



Figure 3.11: Decentralized Replication

Table 3.8: Decentralized Replication Discussion

| **Pros** | **Cons** |
|---|---|
| There is no single master that might become a systems bottleneck. | Different caches are likely to be manipulated by different clients at the same time. |
| Fast when realized with lazy synchronization. | Eager synchronization can avoid data inconsistencies stemming from concurrent updates, but at cost of performance. |
| There is no single point of failure. | Data might become inconsistent with lazy synchronization. |

found in (Özsu & Valduriez, 2011). These protocols are omitted here in order to not losing the focus of the thesis.

This closes the presentation of the state-of-the-art with respect to the approaches to improve the response time of FVPD data. The different approaches, techniques and strategies illustrate the broad variety of opportunities that are offered to address the 2nd hypothesis of this work and choosing adequate strategies for implementing the *caching* approach becomes the major part of the conceptualization, later in this work.

The next section discusses possible benchmarks that are of particular interest in such a distributed scenario before the entire state-of-the-art chapter finishes with a detailed analysis about the current development of the *SeDiCo* framework.

## 3.5 Database Performance Benchmarking

This section analyses different benchmarking frameworks that might be suitable for the performance evaluation of this work. In order to get comparable and reusable results, various current state-of-the-art benchmarks, such as YCSB, SPEC, SPC and TPC are now investigated and the most suitable benchmarking framework is chosen.

First, this section defines the notion of a benchmark in order to reach a common understanding. (Yao & Hevner, 1984) give an appealing explanation of how benchmarking is commonly understood.

*"Benchmarking requires that the systems be implemented so that experiments can be run under similar system environments. [...] In database benchmarking, a system configuration, a database, and a workload to be tested are identified and defined. Then tests are performed and results are measured and analyzed."*

The focus of the following benchmark selection is not on existing bencharking tools, but more on conceptual frameworks that meet the above-mentioned 4 criteria from Gray. Above that, High Performance Computing (HPC) benchmarks as presented in (Akioka & Muraoka, 2010) are not in the scope of this work, as they focus on arithmetic functions such as e.g. LINPACK (Dongarra, 1990). Furthermore, only benchmarks that focus on relational data models are considered as the focus of this work is on current existing (and therefore mostly relational) enterprise databases. Thus, object-oriented database benchmarks, such as e.g. OO7 (Carey et al., 1993) are also out of the scope of this thesis. The same holds for benchmarks that address a specific application domain such as e.g. BG (Ghandeharizadeh & Mutha, 2014) for social networks or RUBIS for auction systems based on middleware architectures (Cecchet et al., 2003). Accordingly, hardware storage focused benchmarks such as SPC (SPC, 2013) or SPEC (SPEC, 2016) are not in the scope of the following analysis.

### 3.5.1 Implications for *SeDiCo*

Based on the just outlined benchmark overview, there remain two benchmarks to be analyzed in more detail: the Transaction Processing Council (TPC) (TPC, 2003) and the Yahoo! Cloud Service Benchmark (YCSB) (Cooper et al., 2010).

Table 3.9 contrasts the two benchmark frameworks (TPC and YCSB) with respect to their applicability in *SeDiCo* (- - very poor, - poor, o neutral, + good, + + very good), and it justifies the usage of the TPC benchmark for the evaluation of the query mechanisms in this thesis.

Table 3.9: Benchmark Discussion

| **Benchmark Criterion** | **YCSB** | **TPC** |
|---|---|---|
| Design | + + | + + |
| Execution | o | + + |
| Analysis | o | + |
| Relevance | + + | + + |
| Portability | + + | + |
| Scalability | + + | + |
| Simplicity | o | + + |
| Relational Database Support | + + | + + |
| Partitioning Support | o | + |
| Cloud Support | + + | + |
| Sum | + + + + + + + + + + + | + + + + + + + + + + + |

This concludes the presentation of the current state-of-the-art and its implications for the entire *SeDiCo* framework[19].

[19]and therefore implicitly for the query mechanisms developed in this thesis

# Chapter 4

# Conceptualization

Generally, there are 3 approaches investigated in this thesis in order to minimize the response time of FVPD data: a *query rewriting*, a *caching*, and an *SSD-based* one. Previously published works of the author can be found in (Kohler, Simov, Fiech, & Specht, 2015)[1] for the *query rewriting* in (Kohler & Specht, 2015c) and in (Kohler & Specht, 2015a) concerning the *caching* approach. The *SSD-based* one has not been published or evaluated so far. These approaches are conceptualized here to optimize the original *SeDiCo* framework implementation outlined in Section 2.

## 4.1 Query Rewriting Approach

The fundamental idea behind this approach is to not only partition and distribute relations and their rows, but also to partition queries accordingly. This section formalizes the entire *query rewriting* approach based on a *projection* issued against two partitions $S_v(B)$ and $T_v(C)$. This *projection* used here is based on two partitions for the sake of better readability.

$$RS(A) \leftarrow \Pi_{(a_1,\ldots,a_n)} R(A) \tag{4.1}$$

[1]This work also demonstrates how additional query filter, join, etc. criteria (previously denoted as $\omega$) are implemented in the *SeDiCo* framework. However, they are ommited here as they are out of the scope and for the sake of better readability.

Note that this initial *projection* is issued against a non-partitioned relation $R(A)$ and the result of this query is written in the result set $RS(A)$. In the next step, a query parser analyses the *projection* to determine which attribute $(a_1, ..., a_i)$ belongs to which partition. Here, it is important to state that the primary key $(a_1)$ is duplicated into both partitions $S_v(B)$ and $T_v(C)$ and the attributes of the projection $(a_1, ..., a_i)$ are matched against the attributes of the partitions $S_v(B)$ and $T_v(C)$.

Thus, according to the definitions from Chapter 1, relation $R(A)$ is a *non-FVPD relation* and the partitions $S_v(B)$ and $T_v(C)$ are the corresponding *FVPD relations*.

Since the partitions are restricted to be *disjoint* and *complete* (cf. Section 1.1), it is ensured that all attributes are matched only once, except for the primary key $(a_1)$ (*disjointness*) and none of the attributes is omitted (*completeness*). After this query parsing, the query is partitioned and issued against the respective partitions, and the result sets with the matching rows are collected:

$$RS_{v1}(B) \leftarrow \Pi_{v1(a_1,...,a_j)} S_v(B) \tag{4.2}$$

$$RS_{v2}(C) \leftarrow \Pi_{v2(a_1,a_{j+1},...,a_n)} T_v(C) \tag{4.3}$$

Then, the result sets have to be joined into the final result set $RS_{final}$:

$$RS_{final} = RS_{v1}(B) \bowtie_{a_1} RS_{v2}(C) \tag{4.4}$$

and due to the *completeness* and *disjointness* criteria it is assured that $RS_{final} = RS(A)$.

A nice advantage of this *query rewriting* is that both projections (4.2 and 4.3) can be run in parallel so that the corresponding result sets $RS_{v1}(B)$ and $RS_{v2}(C)$ can be produced simultaneously.

Listing 4.1: FVPD Nested-Loops Join Algorithm

```
1 for each row r(a1) in R {
2         for each row s(a1) in S {
3                 if (r(a1) = s(a1)) {
4                         put r ⋈a1 s in RSquery
5                 }
6         }
7 }
```

### 4.1.1 FVPD Join

Basically there are 3 types of join algorithms (Graefe, 2011) (Garcia-Molina et al., 2008) (Mishra & Eich, 1992) (Elmasri & Navathe, 2015): *nested-loops join*, *hash join* and *sorted-merge join*. It further has to be noted that previous and current research works extensively examined the challenging performance aspect of vertically partitioned relations. However, all these previous and current works neglect the aspect of FVPD data sets in the context of security and privacy.

In order to improve the performance of the above mentioned approach, this section outlines the three above-mentioned basic join algorithms briefly, relate them to the FVPD approach and present required adaptions of them. These traditional join implementations are used in the thesis to maintain the basic performance evaluation character of the entire *SeDiCo* approach and its previous works, e.g. (Kohler & Specht, 2014a) (Kohler & Specht, 2015a) (Kohler & Specht, 2015c) (Kohler & Specht, 2015b).

**Nested-Loops Join** The nested-loops join algorithm, as its name already suggests, uses two nested loops to collect query-matching rows into a result set:

For every *row* in $R$ and $S$ the join condition is checked and if it matches, the *joined row* is put into the result set. As the two loops (line 1 and 2, Listing 4.1) indicate, the complexity is $\mathcal{O}(|R| * |S|)$ or more general $\mathcal{O}(n^2)$ with respect to the response time[2].

**Hash Join** The hash join is divided into a probe and a hashing phase. Joining two relations $R$ and $S$ is performed according to the following steps:

[2]note that $n$ represents the cardinality of the relations $R$ and $S$ and as they have the same cardinality, it follows that $n = |R| = |S|$, (cf. *completeness*, Chapter 1)

Listing 4.2: FVPD Hash Join Algorithm

```
1  //Phase 1:
2  define hashtable h
3  /* find bigger result set, to hash the smaller one */
4  if (|R| > |S|) {
5          temp = R
6          R = S
7          S = temp
8  }
9  for each row r(a1) in R {
10         put r in h
11 }
12
13 //Phase 2:
14 for each row s(a1) in S {
15         if (s(a1) = h(a1)) {
16                 put r ⋈a1 s in RSquery
17         }
18 }
```

In the first phase (line 4-8, Listing 4.2), the smaller relation (smaller amount of rows) is determined and hashed to save memory[3]. In the second phase, the bigger relation is scanned, hashed against the previously hashed values and matching values are collected in the result set. The complexity (with respect to the response time) of this *hash join* is $\mathcal{O}(hash(R) + |S|)$, which is building the hash table (i.e. scanning relation $R$ once) and check for matching *rows* in $S$. Generally, this can be noted as $\mathcal{O}(n + m)$, or in the presented special case in which $R$ and $S$ have the same cardinality $n$, as $\mathcal{O}(n + n)$.

**Sorted-Merge Join** The *sorted-merge join* is also divided into 2 phases: a *sorting* and a *merging* phase. This algorithm relies on already sorted rows to accelerate the *merging* phase.

In the general *sorted-merge join* the sorting costs are $\mathcal{O}((|R| * log(|R|) + |S| * log(|S|))$ and the merge costs are $\mathcal{O}(|R| + |S|)$, which result in an overall cost of $\mathcal{O}(n * log(n) + m * log(m))$ for the general case. However, the sorting (line 2 and 3, Listing 4.3) in the FVPD approach can be omitted, as the join is performed on the primary key attribute ($a_1$) and this is indexed and therefore already sorted[4].

[3]note that this step can be omitted in this work, as for the evaluation the relations $R$ and $S$ have the same cardinality, (cf. *completeness*, Section 1.1)

[4]note that the steps in line 7-12, Figure 4.3 can also be omitted, as they are only required if the relations have a different cardinality, which is not the case in the FVPD approach (cf. *completeness*, Section 1.1)

Listing 4.3: FVPD Sorted-Merge Join Algorithm

```
1  //Phase 1
2  sort R on r(a1)
3  sort S on s(a1)
4
5  //Phase 2:
6  while (r(a1) in R and s(a1) in S) {
7          while (r(a1) > s(a1)) {
8                  next s(a1)
9          }
10         while (r(a1) < s(a1)) {
11                 next r(a1)
12         }
13         if (r(a1) = s(a1)) {
14                 put r ⋈a1 s in RSquery
15                 next r(a1)
16                 next s(a1)
17         }
18 }
```

Thus, the complexity for the FVPD *sorted-merge join* is (similar to the *hash join*) $\mathcal{O}(|R| + |S|)$ and therefore, $\mathcal{O}(n + n)$, as the relations have the same cardinality $n$.

The complexity of the used join algorithms in query rewriting are summarized in Table 2.1.

Table 4.1: Query Mechanism Complexity

| **Query Mechanism** | **Join Algorithm** | **Complexity** |
|---|---|---|
| Query Rewriting | Nested-Loops Join | $\mathcal{O}(n^2)$ |
| Query Rewriting | Hash Join | $\mathcal{O}(n + m)$ |
| Query Rewriting | Sorted-Merge Join | $\mathcal{O}(n + m)$ |

The next *query mechanism* that is in the focus of this thesis contains three caching approaches, formalized in the following section.

## 4.2 Caching Approach

The three *caching* mechanisms presented in this work can be distinguished as follows:

- Server-Based Caching

These caches are server-based caches (i.e. a cache for every partition, therefore also called decentralized server-based caching) that are operated on different servers between the vertical database partitions and the clients. Every cache only stores tuples from its respective cloud partition and clients access these caches rather than the actual database partitions. Performance improvements are expected from faster access of the cache memory but the actual join of the tuples have to be performed in the clients.

- Local Caching

  This is a cache for each client, as there is a 1:1 connection between client and cache (therefore also called decentralized client-based caching). Here, tuples are already joined (reconstructed) in the cache, which promises performance improvements.

- Remote Caching

  Firstly, it has to be noted that this mechanism violates *SeDiCo*'s *Security-by-Distribution* approach, because a single central server that stores already joined tuples is used (therefore also called centralized server-based caching). However, in order to develop a basic performance metric, this approach is considered useful in the context of this work for the sake of comparability.

The following section outlines the concrete conceptualization of these approaches and how cache coherence is implemented. Therefore, all approaches are formalized and outlined in greater detail, starting with the *server-based caching* approach.

### 4.2.1 Server-Based Caching

Here, every cache stores data from its respective partition. The fact that these caches are on dedicated physical or virtual machines is very appealing, as it might become possible to cache the entire partition and to enhance the overall response time. This could be an approach to avoid the usage of cache synchronization, replacement and invalidation protocols, as all database operations would be directly performed in the caches and the databases would only be used for logging, backup or recovery issues.

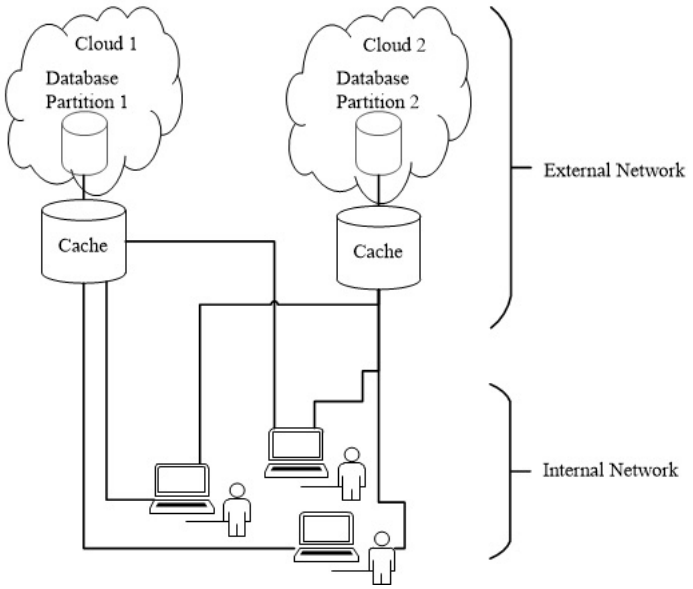



Figure 4.1: Server-Based Caching

Therefore, in this approach cache coherence protocols (cf. Section 3.4.5) are of minor importance, as the caches are used by all clients and data manipulations are directly visible to all of them.

The starting point is a non-partitioned relation $R$ containing attributes $A$: $R(A)$. This relation is also vertically partitioned into partitions $S_v(B)$ and $T_v(C)$, which are *disjoint* and *complete*, accordingly.

This is further illustrated in Figure 4.2.

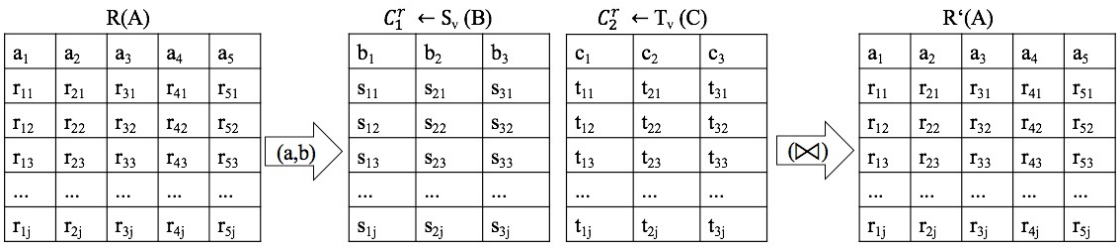



Figure 4.2: Server-Based Caching

In contrast to *local caching*, there are now several decentralized server-based caches $(C_i^r)$[5], in fact, there are implemented as many caches as there are partitions[6]. The projection query to fill the caches (cache warming) is analogous to *local* caching, except that the partitions $S_v(B)$ and $T_v(C)$ are stored in different server-based caches, i.e. $S_v(B)$ in $C_1^r$ and $T_v(C)$ in $C_2^r$.

[5]with C = cache, i = number of the cache, and r = remote
[6]so in the presented approach $i = 2$

### 4.2.2 Local Caching

In this approach, every client has its own cache, which therefore demands *cache synchronization protocols* (i.e. cache synchronization, invalidation and replacement strategies) that ensure data consistency between the FVPD partitions and the caches. This is even more important, as the concrete cache implementation is build on an In-Memory database that uses the Random Access Memory (RAM) of the client as cache memory[7].

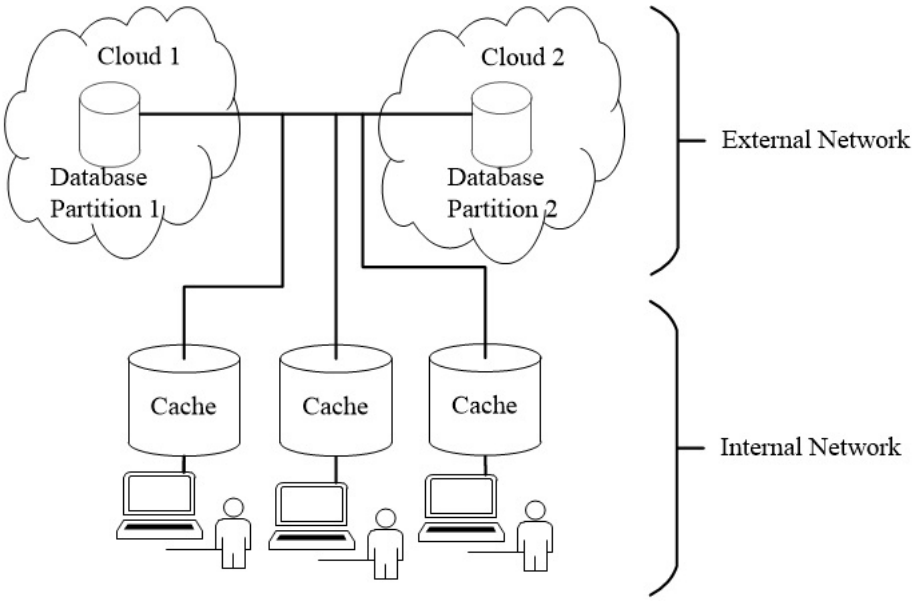



Figure 4.3: Local Caching

Similar to the *server-based* caching approach, the starting point of this approach is a relation $R$ with attributes $A$: $R(A)$. The relation $R(A)$ is then vertically partitioned into 2 partitions $S_v(B)$ and $T_v(C)$. After this partitioning, local client-based caches $(C^l)$[8] come into play. For this, a projection $\Pi$ on both partitions without any filter criteria $\omega$ is performed in order to collect all rows from the respective partitions:

$$RS_{v1}(B) \leftarrow \Pi(S_v(B)) \tag{4.5}$$

[7]especially on e.g. mobile devices but also on current laptops or desktop computers, where the RAM is limited

[8]with C = cache and l = local

and

$$RS_{v2}(C) \leftarrow \Pi(T_v(C)) \tag{4.6}$$

After that, the two result sets $RS_v(B)$ and $RS_v(C)$ are joined to the final result set $RS_{final}$ and due to the fact that no filter criteria $\omega$ were used, all attributes were selected and all rows of $RS_v(B)$ and $RS_v(C)$.

Then, $RS_{final}$ is stored in the *local cache* ($C^l$):

$$C^l \leftarrow RS_{final} \tag{4.7}$$

These steps are also illustrated in Figure 4.4.

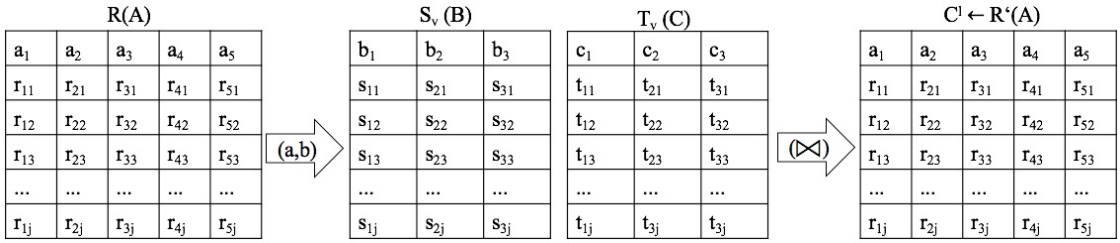



Figure 4.4: Local Caching

As soon as the local cache ($C^l$) contains all rows from (4.7), recall that *completeness* and *disjointness* (cf. Section 1.1) are also prerequisites for this approach, all queries are issued exclusively against the cache ($C^l$). On the contrary, data manipulations are only permitted directly on the FVPD partitions in the context of this work. As soon as a row is modified, the modification is propagated to all involved caches, i.e. with a so-called write-through strategy[9]. For this, a time-based synchronization interval (configurable in milliseconds) between the FVPD partitions and the caches is used.

In the end, this approach demonstrates that the overall response time is heavily dependent on the concrete cache implementation (e.g. In-Memory, file-based, etc.), on the storage capacity of the cache and on the used cache coherence protocols, i.e. synchronization, invalidation and replacement (cf. Section 3.4). This thesis uses a *master slave replication* (Figure 3.10), as the advantages of this approach outweigh

[9]In this strategy, all manipulations in the cache are directly propagated to the respective FVPD partitions. This is completely transparent for the client, i.e. it does not know whether it actually operates on a cache or on FVPD data.

the cons[10]. Namely, the clients do not need any kind of synchronization logic and it is ensured through the *session* concept of Hibernate and the write-through strategy, that the cloud partitions have the most current data and these are propagated to the respective client-side caches. Another caching approach that is addressed in this thesis is remote caching, which is conceptualized in the following section.

### 4.2.3 Remote Caching

Figure 4.5 illustrates the *remote caching* approach.

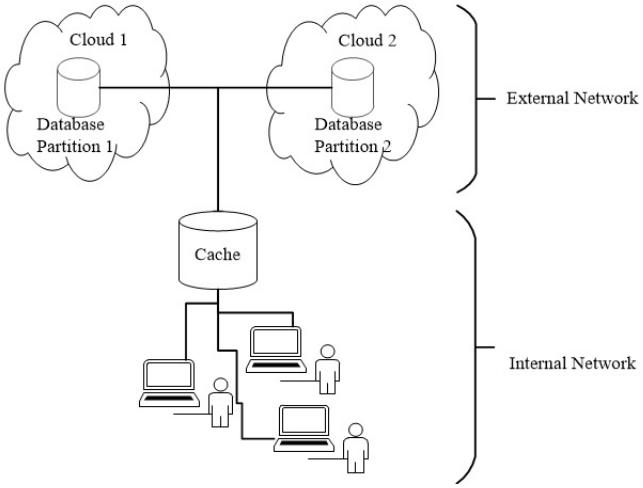



Figure 4.5: Remote Caching

This approach is similar to the *local caching* approach, except that here a single centralized server-based cache ($C^r$)[11] is used. All clients operate on this cache in the same manner as they operate on a *decentralized client-based cache*. Therefore, this approach is not conceptualized here again. Advantageous of this approach is the possibility to use a larger cache memory, more processors, etc. due to the larger hardware dimensions of a server. Contrarily, the violation of the *Security-by-Distribution* principle is a great disadvantage if this server-based cache is not in a secure network, only accessible via e.g. virtual private network (VPN).

This closes the conceptualization of the caching approach and moves the SSD-based approach into consideration in the following section.

[10] even the cons of the decentralized replication (Figure 3.11

[11] with C = cache and r = remote

## 4.3 SSD-Based Approach

The SSD-based approach is similar to the original *SeDiCo* approach, depicted in Section 2.2 and therefore, its conceptualization is outlined very briefly in this section. Figure 4.6 illustrates the entire architecture to provide a better overview about the initial FVPD approach.

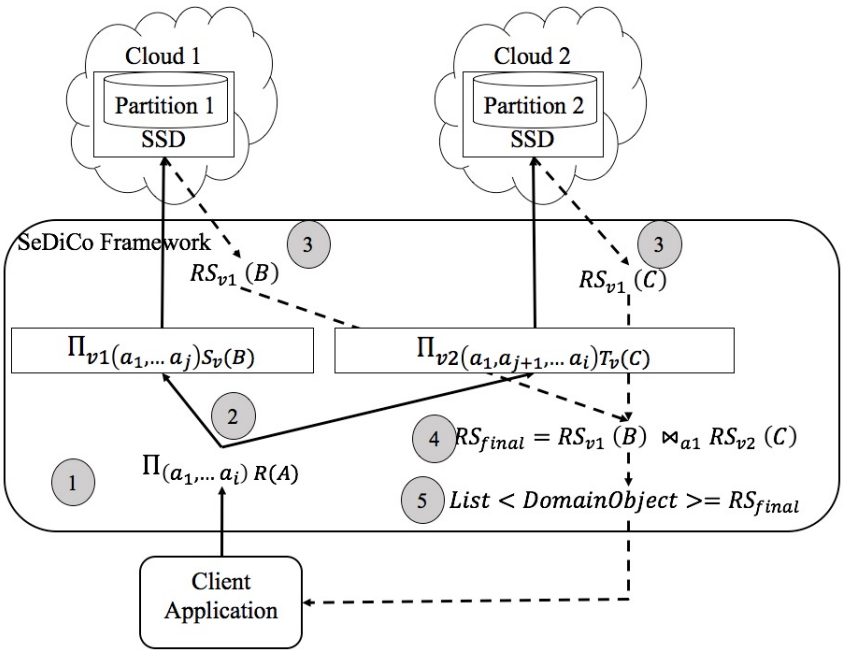



Figure 4.6: SSD-Based Architecture

Basically, the FVPD approach (cf. Section 2.2) is applied, which can be described as follows: a relation $R(A)$ is vertically partitioned into partitions $S_v(B)$ and $T_v(C)$ and these partitions are then distributed across different clouds.

1. A query $\Pi_{(a_1,\ldots,a_i)}R(A)$ issued against the original database.
2. The query is rewritten to fit into the FVPD scheme and its reconstruction queries $\Pi_{v1(a_1,\ldots,a_j)}S_v(B)$ and $\Pi_{v2(a_1,a_{j+1},\ldots,a_i)}T_v(C)$ are issued against the respective partitions $S_v(B)$ and $T_v(C)$.
3. After that, the result sets of the partitions ($RS_{v1}(B)$ and $RS_{v2}(C)$) are collected.
4. Then the result sets $RS_{v1}(B)$ and $RS_{v2}(C)$ are joined via natural joins on their primary key attribute $a_1$: $RS_{final} = RS_{v1} \bowtie_{a_1} RS_{v2}$.
5. The result set $RS_{final}$ is mapped to a list of domain objects and delivered to the querying client.

6. An additional *optional* step is required if the original query $\Pi_{(a_1,\ldots,a_i)}R(A)$ contains projection criteria $\omega$ that indicate with attributes should be part of the result set. In that case, these criteria have to be applied on the respective object attributes of the previously created list and query matching objects are kept in the list, whereas other objects are removed.

The fundamental idea of this approach is that a major performance gain concerning the collection of the result sets and the join performance can be achieved with the usage of SSD drives that store the respective partitions.

The complexity of the SSD-based approach is equal to the initial FVPD approach, as no algorithmic optimization is performed. It is summarized in Table 4.2.

Table 4.2: Query Mechanism Complexity

| **Query Mechanism** | **Join Algorithm** | **Complexity** |
|---|---|---|
| SSD-based | Nested-Loops Join | $\mathcal{O}(n^2)$ |

This concludes the conceptualization of the query mechanisms and based on these architectural overviews, the following chapter outlines their concrete implementation.

# Chapter 5

# Implementation

With respect to the formal description of the *query rewriting*, the *caching*, and the *SSD-based* query mechanisms, this chapter now outlines their concrete implementation. Figure 5.1 gives an overview about the location of the respective mechanisms and their integration into the *SeDiCo* framework. Hence, Figure 5.1 is used as an overview about the structure of this chapter which firstly outlines the concrete *query rewriting* implementation, secondly, the *caching* and lastly the *SSD-based* approach.

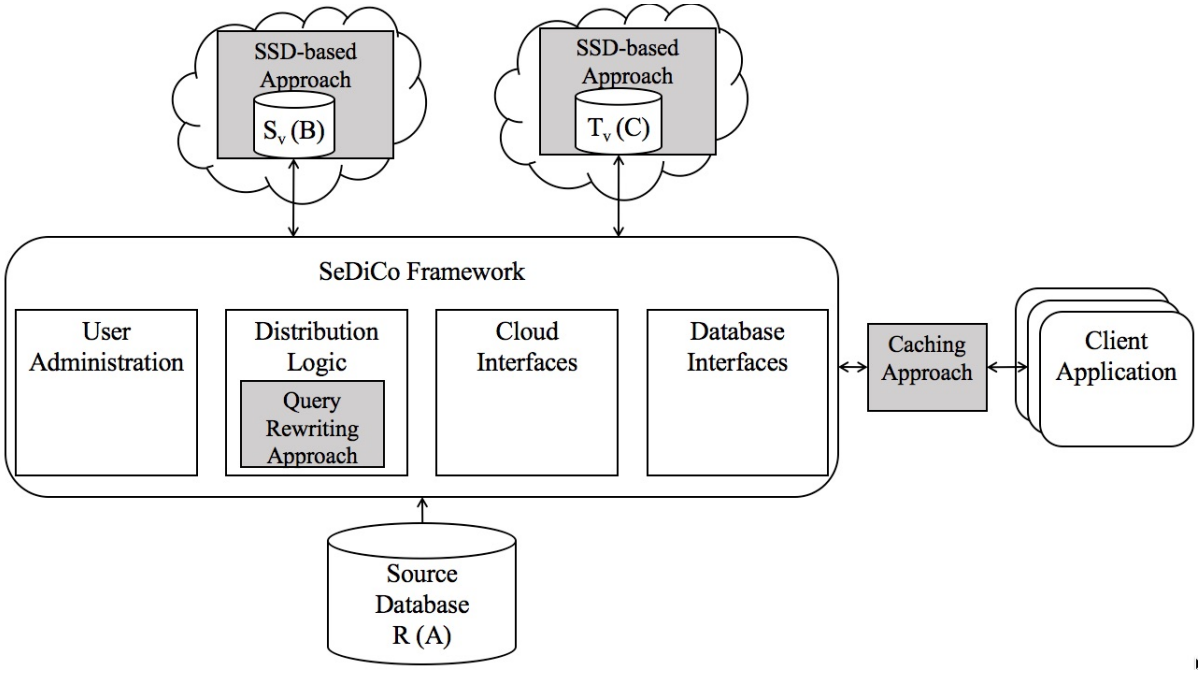



Figure 5.1: *SeDiCo* Query Mechanism Integration Overview

In order to give a comprehensible description of the query strategies, the Unified Modeling Language (UML) is used. UML provides a readily understandable and

standardized (ISO/IEC, 2005) way how algorithms, software components, etc. can be described. Here, a total of 14 different UML diagram types (e.g. class, use case, activity, sequence diagrams, etc.), depending on the view of an algorithm, a software component, etc. can be used. In the context of this chapter, the UML *sequence diagram* provides the most suitable way to present the interaction of the different components in the respective query strategy, as it models the behavior of the respective components. Hence, they are used here to illustrate the concrete implementation of the respective strategy. For an exhaustive overview about all UML diagrams the reader's attention is drawn to the relevant literature, e.g. (Booch et al., 2005) (Omg, 2011).

Fig. 5.2 gives an overview about the key concepts of UML *sequence diagrams*, to create a common understanding for the following *sequence diagrams* in this chapter.

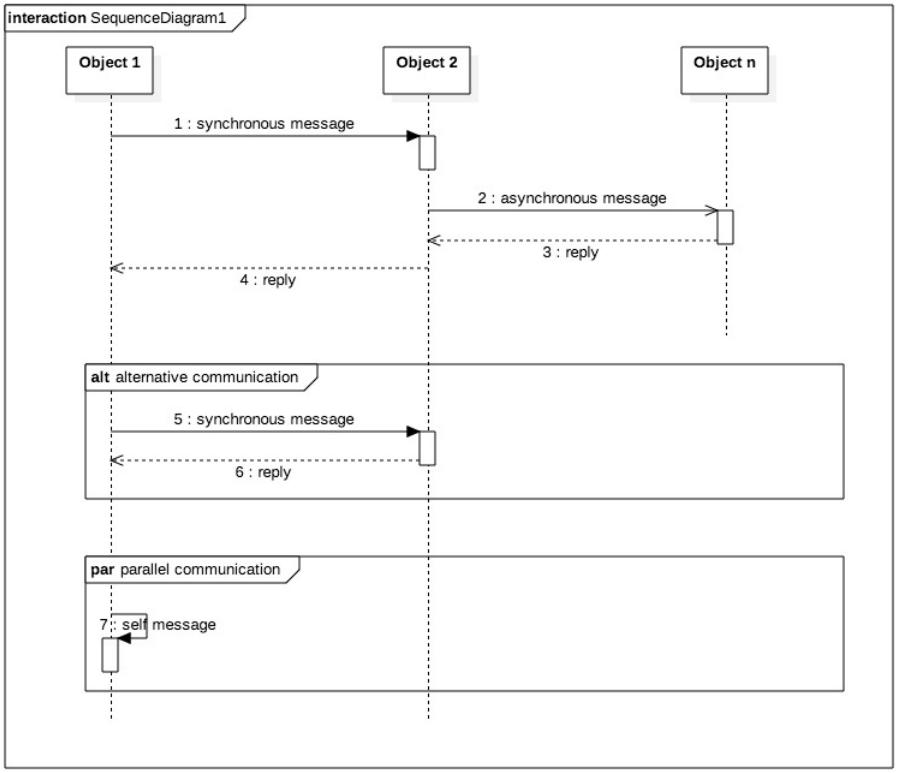



Figure 5.2: UML Sequence Diagram Key Concepts

Here, different objects (software components, persons, etc. could also be used), e.g. in an object-oriented program communicate with each other via messages. The objects send messages and replies to each other to perform a certain task. Time is running from top to bottom and the ordering of the messages is depicted

with the preceding number in the front of each message. Above that, the dotted line of each object illustrates its lifeline. As messages 1 and 2 show, sending and receiving messages can be done synchronously or asynchronously. Another important concept is depicted with the *alt* and *par* fragment which state that the communication within these fragments takes place alternatively[1] or simultaneously in parallel[2]. In addition to this, there are other fragments (i.e. *opt* for an optional communication, or *loop* for an repeating loop construct, *assert* for a communication that is mandatory, etc.). A more detailed introduction can be found in (Booch et al., 2005) (Omg, 2011), as here only relevant concepts for the thesis are described.

## 5.1 Query Rewriting Implementation

As already stated in Section 2.2.1, joining query-matching rows from corresponding result sets based on their primary keys is a key element for the performance evaluation of the *query rewriting* approach. The following section picks up the 3 basic join algorithms (nested loops, hash and sorted-merge join) and adapts them to the *SeDiCo* approach, but firstly, the optimized query rewriting strategy is outlined in more detail.

The initial *SeDiCo query rewriting approach*, depicted in Section 2.2.2, is now advanced by more sophisticated join algorithms. The initial idea of an XML-based mapping of the query attributes to their respective partitions is maintained. In the end, this requires thread synchronization before the natural join can take place. The UML sequence diagram depicted in Figure 5.3[3] illustrates this in more detail.

In contrast to the initial *query rewriting approach* of *SeDiCo*, messages[4] 3-10 are parallelized. Thus, the performance gain is expected through the parallel query execution and the parallel join of the respective result sets. Figure 5.3 also shows that the collection and transfer of the result sets (message 6 and 10) requires synchronization for all collecting threads, as message 11 can only be performed unless all intermediate result sets have entirely been collected. Otherwise, the final result set would be incomplete.

[1] wich would be implemented as an if-else block in a programming language

[2] which would be implemented with different threads in a programming language

[3] note that the indices e.g. $(a_1, ..., a_i)$ in all Figures in this chapter are depicted as (a_1, ..., a_i)

[4] in UML the respective *steps*, depicted in the figures are called *messages*

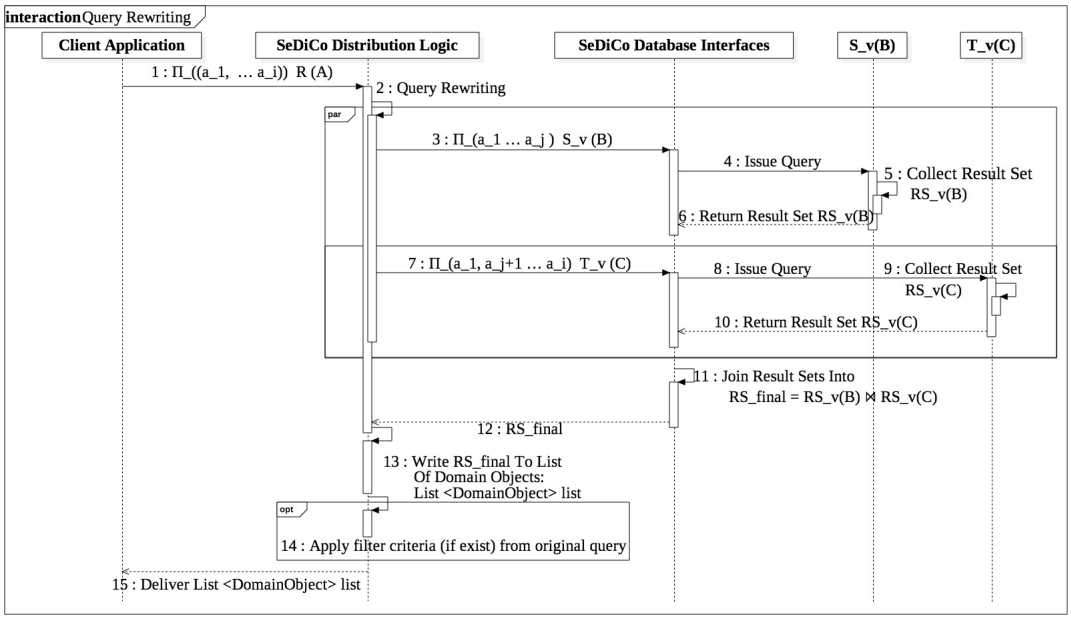



Figure 5.3: Query Rewriting Implementation

As the query rewriting is now completely outlined, a closer look is taken into message 11, which contains the *natural join* of the collected result sets.

### 5.1.1 FVPD Join Implementation

The original relation $R(A)$ is vertically partitioned into $S_v(B)$ and $T_v(C)$. Now, a *natural join* on the replicated primary key attribute $(a_1)$ is used to reconstruct rows from the original relation $R(A)$:

$$R(A) = S_v(B) \bowtie_{a_1} T_v(C) \tag{5.1}$$

In order to perform this join, the 3 basic join algorithms (*nested-loops*, *hash* and *sorted-merge*) are implemented and experimentally evaluated.

To sum it up, the evaluation of this approach will show the performance gain of all 3 mentioned join algorithms, combined with the performance of the query rewriting algorithm.

All in all, messages 12-14 (Figure 5.3) finalize the concrete implementation and delivers the result set to the querying Java client in form of a list of objects[5]. In more detail: after the result set $RS_{final}$ is written into the list of *domain objects* (message 13), the *original query* has to be issued against this list once more (message 14), which ensures that the complex combination of filter criteria (i.e. query attributes) is maintained for the entire query.

## 5.2 Caching Implementation

This section presents the *caching* approach with its three variations: the *server-based* and *local* and the *remote* approach. Analogous to the previous sections, this section takes a closer look on their concrete implementations which are subsumed in the *caching* approach. This section starts with an introduction of the used cache coherence protocols and with further common features of all components. At first, the *server-based* caching implementation and an overview about its evaluation goals are described. Afterwards, the *local* and the *remote* approaches are outlined in more detail.

**Cache Coherence Protocols** With respect to caching, there are two main differences concerning the *cache coherence protocols*: firstly, the *synchronization* between the caches and the FVPD partitions and secondly, the *synchronization* between all the caches. The first synchronization challenge is implemented with a time interval-based synchronization, whereas the latter one is implemented under the restriction that row modifications (DML operations) are only performable directly on the FVPD partitions and never on the caches. This restriction facilitates the time-based synchronization and easily removes cache coherence challenges.

**Cache Memory** A further question concerning *caching* is the implementation of the cache memory. Here, the author distinguished between four different cache memory implementations in (Kohler, Simov, Fiech, & Specht, 2015): a *key-value based In-Memory cache*, a c*lient-side relational database* used as a cache and two *file-based caching* solutions, also located at the client that use a JSON file as cache memory. The author showed in an experimental setup which is also

[5]because of the Impedance Mismatch between the relational and the object-oriented paradigms, outlined in Chapter 3

used in this thesis, that the In-Memory cache outperforms the other approaches averagely, if the data set fits entirely into the cache[6]. However, the usage of In-Memory databases as caches involves drawbacks due to the volatility of the storage. As soon as the cache is powered off, all cached data are lost and have to be re-cached again and this is of particular importance in client-based cache implementations, as they are turned off and on more often than server-based ones.

Another issue concerns the storage capacity of the cache implementations. Although In-Memory caches are the fastest ones compared to the other evaluated ones, RAM storage is more expensive than HDD or SSD storage. Nevertheless, as this thesis focuses on the performance of the respective approaches, the fastest one is evaluated in this section. To provide an overview about all four cache memory implementations, their respective response times are presented in Figure 5.4.

These results are discussed in more detail in (Kohler, Simov, Fiech, & Specht, 2015), thus this section only mentions the key characteristics to understand the results. Figure 5.4 shows the performance of a key-value based In-Memory store, a MySQL database locally installed on the client as a cache and a file-based JSON cache. There are 2 evaluations for the file-based JSON cache: in the first case, the cache file contains all 88K already joined rows from the FVPD partitions (depicted as *json_full*). In contrast to this, the cache file in the *json* approach contains the respective number of rows in the cache that were actually queried. As the cache file is smaller, reading the entire file is faster than in the prior approach.

[6] which is more likely in server-based caching approaches due to larger hardware dimensions (i.e. cache memory, processor speed, etc.)

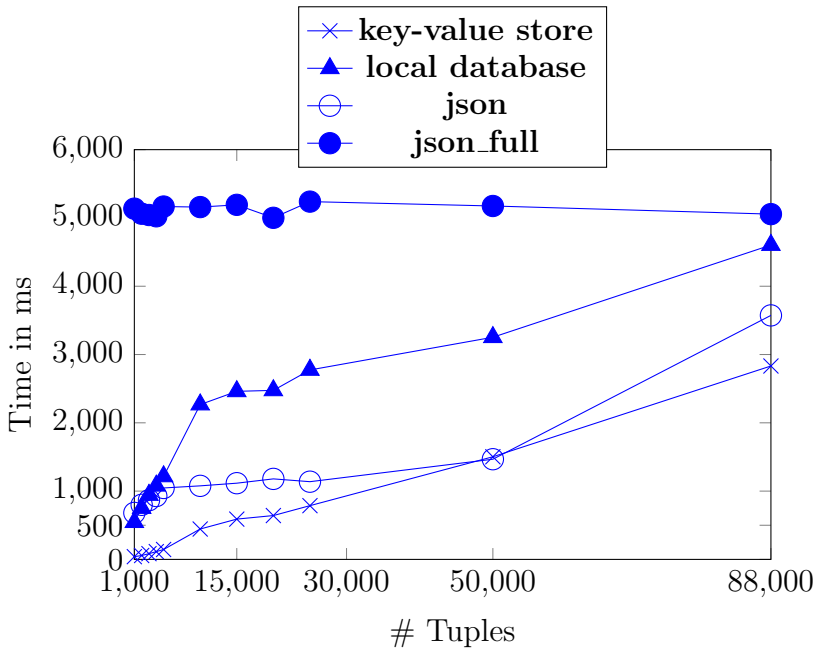



Figure 5.4: Cache Performance Comparison

The results show that the key-value based In-Memory cache is the fastest one, if only the response time is considered and the following caching approaches are all evaluated with such key-value based In-Memory caches.

### 5.2.1 Server-Based Caching

The first message in this procedure is to fully initialize all caches (cache warming, messages 1-8). As there is one dedicated cache for each FVPD partition, no join is performed at that time. It has to be noted that there are various cache synchronization strategies possible in this scenario. Figure 5.5 shows a time interval-based approach (messages 1-8), which is implemented with a configurable time property in milliseconds.

Finding adequate synchronization time intervals is heavily dependent on the database workload. As the goal of this thesis is the evaluation of a basic response time, synchronization as well as replication, invalidation and replacement strategies are not further followed. Due to the distribution of the entire data set across various caches, cache misses are expected to be reduced compared to centralized caches that are not able store the entire data set. This will be investigated in future works in order to not loosing the focus of this thesis, which is the basic evaluation of query strategies without external influences like minimal cache memory (i.e.

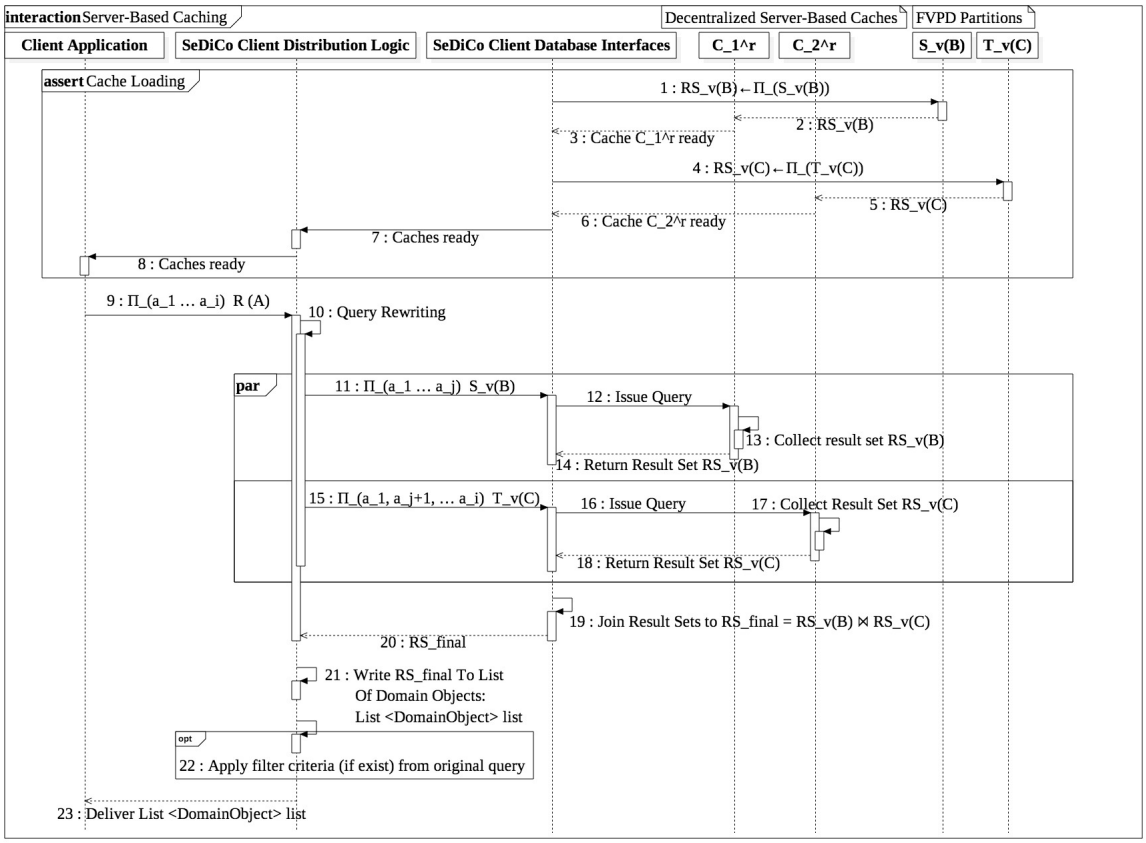



Figure 5.5: Server-Based Caching Implementation

cache hits and misses) or workload-driven access patterns (i.e. OLTP versus OLAP).

As the caches do not store entirely reconstructed rows unlike in the following *local* caching approach, the original client query must also be rewritten according to the *SeDiCo* approach outlined in Section 2.2 and this is illustrated in messages 9-20 (Figure 5.5). Another advantage of the integration of *server-based* caches for the FVPD partitions, is that the caches can also be located in even public clouds, as they only store the respective FVPD chunks. As data volumes grow, the cache memory would then easily be dynamically scalable. After the result set $RS_{final}$ is written into the list of *domain objects* (message 21), the *original query* has to be issued against this list once more (message 22), which ensures that the complex combination of filter criteria (i.e. query attributes) is maintained for the entire query.

Performance improvements are expected from this cache implementation, as rows do not have to be fetched from the FVPD partitions, but from faster cache

memories. This is assumed to reduce database I/O as the entire data set is cached in various caches.

### 5.2.2 Local Caching

In contrast to the *server-based* implementation, the *local* one suffers from smaller hardware dimensions with respect to cache memory. The clients in this setup do not share the cache memory, every client owns its specific cache and therefore the required cache memory can be reduced. However, a reduced cache memory requires advanced cache loading and synchronization strategies to minimize cache misses. As the data set is not modified for the evaluation in this work, cache loading, synchronization and replacement strategies are not in the main focus, but they become important in future work challenges that deal with an efficient usage of a limited cache memory.

Figure 5.6 illustrates the entire local caching approach in form of an UML sequence diagram.

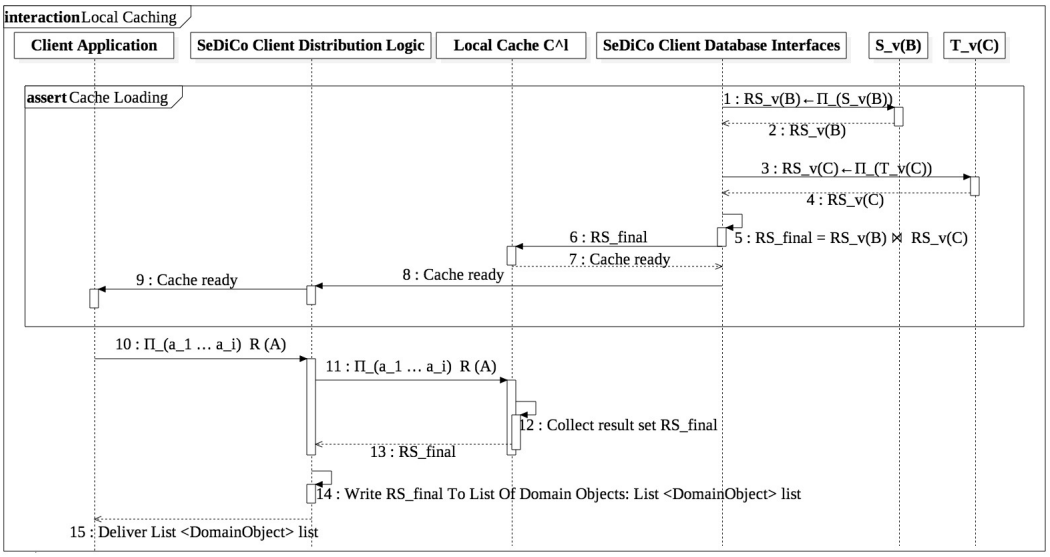



Figure 5.6: Local Caching Implementation

Answering a query is depicted in messages 10-15 and the cache loading (i.e. warming) phase is illustrated in messages 1-9 (Figure 5.6[7]). As the local cache stores already reconstructed (i.e. joined) tuples (message 5), no query rewriting has to take place in messages 10-15 (Figure 5.6), as the original query is issued directly against the cache.

[7] with ($C^l$) denoted as (C^l)

### 5.2.3 Remote Caching

As already illustrated in Section 4.2.3, the *remote* approach consists of a comparatively insecure *server-based cache* implementation[8]. Basically, the process of answering a query in this approach is illustrated in messages 10-15 (Figure 5.7[9]), which is similar to the local caching approach depicted in Figure 5.6. An advantage of this cache implementation and the above-mentioned *server-based* one, is the fact that caching servers, especially those based on a dynamically scalable cloud infrastructure can be dimensioned to cache the entire data set of the FVPD data. Further application scenarios for the usage of FVPD data could be logging, backup and recovery or high availability setups. Another advantage is that no client modifications are necessary in these implementations, as various clients share the same common cache.

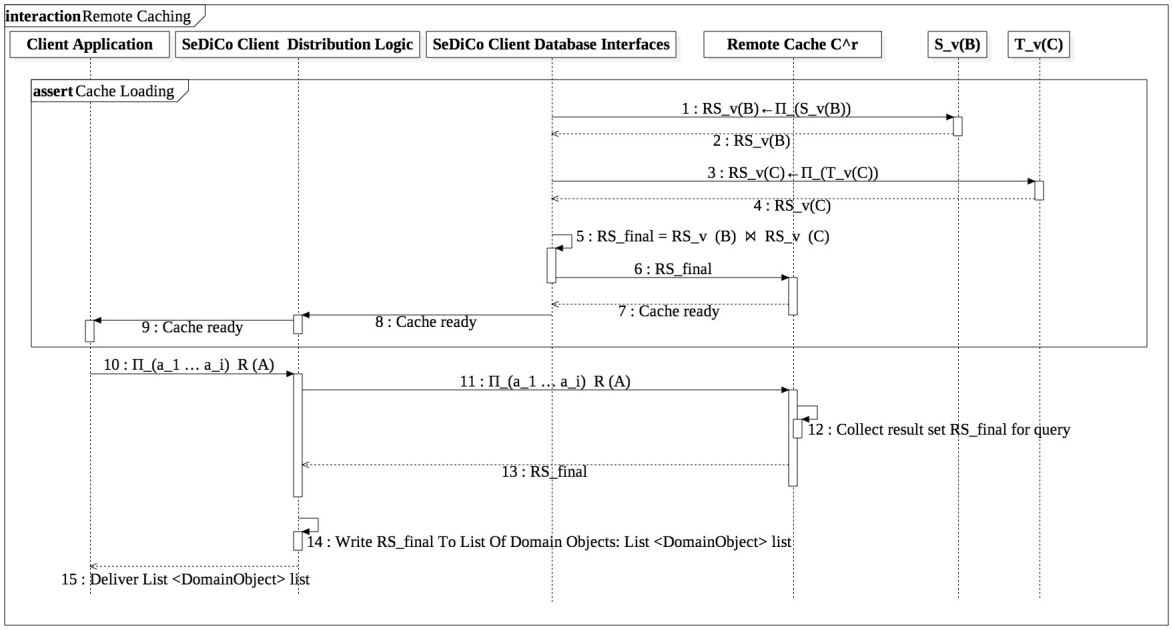



Figure 5.7: Remote Caching Implementation

In the end, the evaluation in the next chapter will show, how the performance differs in the *server-based*, in the *local*, and in the *remote* approaches. As no modifications in the data sets are performed, the pure cache performance is evaluated and this serves as a basis for further works that will include cache synchronization, replication, invalidation and replacement strategies.

[8] because the tuples are stored already joined in the external remote cache

[9] with ($C^r$) denoted as (C^r)

## 5.3 SSD-Based Implementation

This implementation has a strong empirical character, as no new algorithms or query strategies are developed. Here, the influence of new hardware capabilities in form of Solid State Drives (SSD) is measured and this will show to which extent new technological developments are able to improve the FVPD approach.

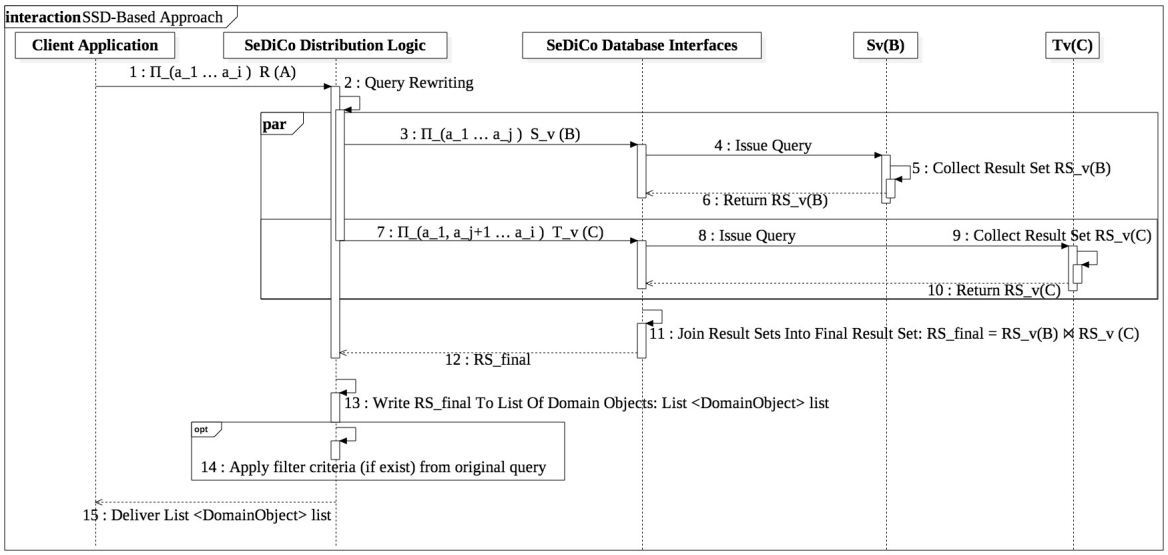



Figure 5.8: SSD-Based Implementation

As illustrated in Figure 5.1, the key concept of this approach is the usage of SSDs as secondary storage for the database partitions in their corresponding clouds. The UML sequence diagram in Figure 5.8 outlines the implementation in more detail and illustrates the interplay of the involved components.

The 15 messages in Figure 5.8 illustrate the entire *SeDiCo* approach in its concrete implementation.

In summary, the evaluation of the *SSD-based* approach compares the analysis of the response time to the initial *SeDiCo* implementation and to the other query strategies in order to determine the performance gain.

# Chapter 6

# Evaluation

This chapter now covers the evaluation of the 3 previously described query mechanisms. Firstly, the evaluation environment, i.e. the data set and structure, the entire hardware environment and the database management systems used to measure the response time of the query mechanisms are described. Secondly, a basic performance metric is developed in order to compare the response time against non-partitioned and non-distributed settings (Section 6.2) and against the initial *SeDiCo* approach (Section 6.3). Then, all query mechanisms are evaluated in the rest of this chapter. At the end, there is a conclusion that summarizes the respective results. A final summary of the main results that compares and interprets all results can then be found in Chapter 7.

## 6.1 Evaluation Environment

The data set for the evaluation is derived from the *CUSTOMER* table of the TPC-W benchmark (TPC, 2003). This *CUSTOMER* table is partitioned and distributed according to Figure 6.1.

R(A)

| CUSTOMER | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_PASSWD | VARCHAR NOT NULL |
| | C_UNAME | VARCHAR NOT NULL |
| | C_FNAME | VARCHAR NOT NULL |
| | C_LNAME | VARCHAR NOT NULL |
| | C_ADDR_ID | INT NOT NULL |
| | C_PHONE | VARCHAR NOT NULL |
| | C_EMAIL | VARCHAR NOT NULL |
| | C_SINCE | DATE NOT NULL |
| | C_LAST_LOGIN | DATE NOT NULL |
| | C_EXPIRATION | TIMESTAMP NOT NULL |
| | C_DISCOUNT | DOUBLE NOT NULL |
| | C_BALANCE | DOUBLE NOT NULL |
| | C_YTD_PMT | DOUBLE NOT NULL |
| | C_BIRTHDATE | DATE NOT NULL |
| | C_DATA | VARCHAR NOT NULL |

Sv (B)

| CUSTOMER_p1 | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_PASSWD | VARCHAR NOT NULL |
| | C_UNAME | VARCHAR NOT NULL |
| | C_FNAME | VARCHAR NOT NULL |
| | C_LNAME | VARCHAR NOT NULL |
| | C_ADDR_ID | INT NOT NULL |
| | C_PHONE | VARCHAR NOT NULL |
| | C_EMAIL | VARCHAR NOT NULL |
| | C_SINCE | DATE NOT NULL |

Tv (C)

| CUSTOMER_p2 | | |
|---|---|---|
| PK | C_ID | INT NOT NULL |
| | C_LAST_LOGIN | DATE NOT NULL |
| | C_EXPIRATION | TIMESTAMP NOT NULL |
| | C_DISCOUNT | DOUBLE NOT NULL |
| | C_BALANCE | DOUBLE NOT NULL |
| | C_YTD_PMT | DOUBLE NOT NULL |
| | C_BIRTHDATE | DATE NOT NULL |
| | C_DATA | VARCHAR NOT NULL |

Figure 6.1: FVPD TPC-W *CUSTOMER* Table

In order to achieve a better comparability throughout all previous works of the author and this thesis, all evaluations are performed with a data set that ranges from 0 to 288K randomly generated rows, which result in an overall database size of:

| **Table** | **Size in MB** |
|---|---|
| CUSTOMER ($R(A)$) | 147 |

Table 6.1: Data Set Size of Relation $R(A)$

| **Tables** | **Size in MB** |
|---|---|
| CUSTOMER_p1 ($S_v(B)$) | 56 |
| CUSTOMER_p2 ($T_v(C)$) | 113 |

Table 6.2: Data Set Size of Vertical Partitions $S_v(B)$ and $T_v(C)$

In this scenario, the sum of $S_v(B)$ and $T_v(C)$ (169 MB) is greater than the size of the original table ($R(A)$). This refers to the duplication of the primary keys ($a_1$) into both partitions, which is 11 MB per partition (169 MB − 147 MB = 22 MB for both partitions). Other optimization techniques such as indices or local database caches are not used due to their locally focused optimization scope.

It has to be mentioned that the figures in this section are only excerpts of the evaluation because of the great variation in the response times from 1 to 288K tuples. Hence, the figures only illustrate the response times from 1K to 88K tuples, which is considered the best trade-off between informational value and readability.

**Local Evaluation Environment** All performance measurements are conducted on a single physical machine with the following hardware dimensions:

- CPU: 2.4 GHz AMD Dual Core
- RAM: 8 GB
- SSD/HDD: 250 GB
- Software: CentOS 6, Java 1.7_79 64Bit, MySQL 5.6, Oracle Express 11g

On first sight, this *local* setup contradicts the distributed cloud computing scope of this work, but it has the advantage of comparable and reproducible measurements, due to the following reasons:

- In a distributed Cloud Computing environment, a virtualized infrastructure based on physical hardware is used as a basic technology in order to abstract from the concrete physical systems. Due to this so-called multi-tenancy setup where several cloud users share the same physical resources, separated through a virtualization layer, the overall utilization of the underlying physical systems is optimized. However, this additional virtualization layer decreases the performance by $\sim 7\%$, as (Grund et al., 2010) showed in their study.
- Closely related to these virtualization issues, is the challenging reproducibility of performance measurements in such an environment. Here, the problem of the unknown overall utilization (esp. in public clouds) comes into play. Consider a setup with two virtual machines $m_1$ and $m_2$ in a virtualized cloud on the same physical machine that use the entire available physical resources where $m_1$ is used for an evaluation task and $m_2$ is an unknown machine that belongs to someone unknown. At time $t_0$ the utilization of $m_1$ is 90% and $m_2$ has 10%. So, the overall utilization of the physical hardware is 100%[1]. Then later at time $t_1$ the same evaluation task is performed on

[1]the hypervisor and other virtualization overhead is neglected here for the sake of clarity

$m_1$, but this time the utilization of $m_2$ is 90%, and there are only 10% left for $m_1$. Both, the overall utilization and the utilization of $m_2$ cannot be influenced by the owner of $m_1$ and transferred to a public cloud with various virtualized resources and no possibility to influence or monitor the overall utilization of the underlying physical hardware systems, such a setup is regarded inappropriate for comparable and reproducible measurements. Moreover, the usage of the same physical hardware systems, only divided by a software-based virtualization layer (which may include bugs), can be regarded as insecure with respect to data privacy and security.

All in all, it can be concluded that the latter two arguments could be solvable with *Service Level Agreements* (a detailed consideration can be found in (Kohler & Specht, 2014c)) and a precise definition of the rented cloud capabilities (e.g. CPU-time, RAM, HDD, network bandwidth, etc.). However, besides the cost-intensive and time-consuming definition, the network overhead would be another challenge, as all rows would have to be transferred via Internet and its unpredictable network bandwidth behavior to and from the cloud providers. Another consideration would be the usage of a private cloud setup, which is described in more detail in the following section.

**Remote Evaluation Environment** The unknown overall utilization, the virtualization overhead and the network overhead lead to the local setup on 1 physical machine as described above to achieve a basic performance metric. It further has to be noted that the initial response time in (Kohler & Specht, 2014a) contains a *local* and a *remote* evaluation. This *remote* evaluation was performed in a private cloud environment, to which only the author of this thesis had access to. This private cloud infrastructure was built on the following hardware:

- CPU: 2.4 GHz AMD Dual Core
- RAM: 2 GB
- SSD/HDD: 250 GB
- Network: 1Gbit
- Software: CentOS 6, Java 1.7_79 64Bit, MySQL 5.6, Oracle Express 11g

Based on this these capabilities, two private cloud infrastructures (Eucalyptus 3 (Hewlett Packard, 2016) and CloudStack 3.2.2 (Apache, 2016a)) with à 5 computing nodes (1 cloud management server and 4 cloud nodes) were set up. Thus, the above-mentioned cloud challenges could be avoided. Accordingly, the evaluation results in (Kohler & Specht, 2014a) show that the *remote* setup did have remarkable impacts compared to the *local* setup, which is averagely $\sim$ 60%-$\sim$ 90% faster.

It also has to be noted that the Oracle database system is remarkably slower (with datavolumes $\geq$ 50K tuples) compared to the MySQL database. This is because of the used Oracle Express Edition 11g Release 2, which is restricted to the usage of 1 processor and 1 GB RAM (Oracle, 2016) in its publically available version. Contrarily, MySQL is not restricted at all, but recent evaluations with other database systems (MariaDB, PostgreSQL) show similar results. To remain comparable to the other previously published works (listed at the end of the thesis), this evaluation continues with the measurement of MySQL and Oracle databases. Above that, the original motivation for those two systems was their wide distribution in industrial environments.

It further has to be noted that the following performance evaluations also include the measurements of a simultaneous usage of both database systems. This is the so-called *combined response time*. As the *SeDiCo* framework offers the possibility to use both database systems for partitions at the same time, it is possible to store one FVPD partition in an Oracle and the other in a MySQL database (or vice versa).

## 6.2 Basic Database Performance Evaluation

This initial performance measurement serves as a basic performance metric for all following evaluations. All performance measurements were conducted three times and the average times of all measurements are listed in the figures of this chapter. With this approach, unreproducible side-effects like Java's Garbage Collector or changing host utilization could be reduced to a minimum.

Figure 6.2[2] presents the response time of a *locally* and *remotely* installed, non-partitioned and non-distributed database. As the data set is neither partitioned

[2]cf. (Kohler & Specht, 2014b)

nor distributed, it basically measures the pure response time of the ORM (i.e. Hibernate). So, the hypotheses (cf. Section 1.4) can be confirmed, if the average response time for the FVPD data is equal or even smaller compared to the average response times of Figure 6.2.

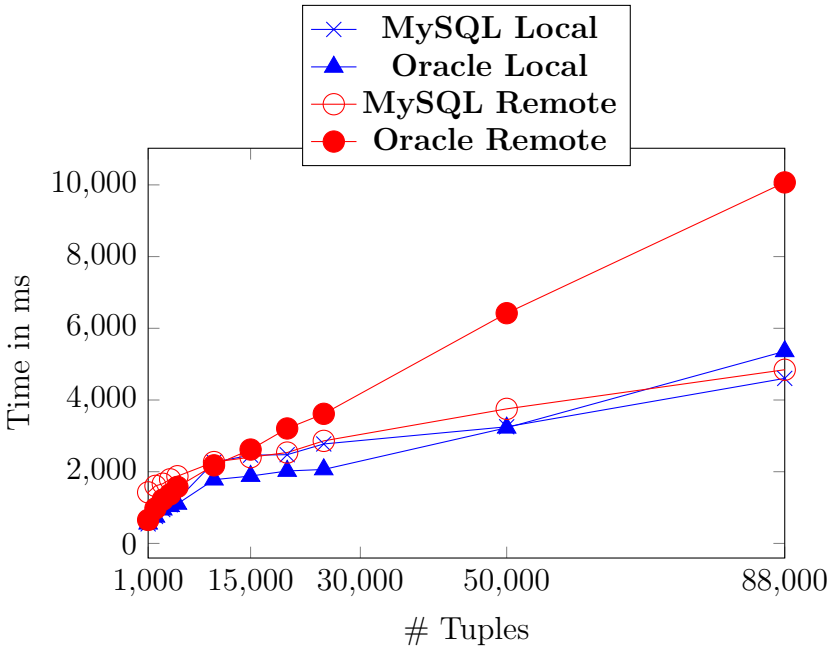



Figure 6.2: Initial Response Time

### 6.2.1 Conclusion

Considering the average response time of Hibernate, based on a non-distributed and non-partitioned data set, these results show that querying a MySQL database requires averagely ∼1,6 seconds and an Oracle database which is nearly similar requires ∼1,7 seconds. These values stem from a *local* environment where all components are installed on one single physical machine. Although this is not applicable in real world scenarios, these figures provide a basic performance metric. In a *remote* setup (i.e. a client-server environment), querying a MySQL database requires ∼2 seconds and an Oracle database needs ∼3 seconds to answer the query averagely. Hence, it can be concluded that the network overhead is ∼0.3 seconds (MySQL) and ∼1,4 seconds (Oracle) and this is insofar interesting as the following performance measurements will show, how this network overhead affects the *Security-by-Distribution* approach of the *SeDiCo* framework.

## 6.3 *SeDiCo* Framework Performance Evaluation

Similar to the previous section, this evaluation is also divided into a *local* and a *remote* measurement and the following sections present the response time of the initial *SeDiCo Security-by-Distribution* approach without any optimizations.

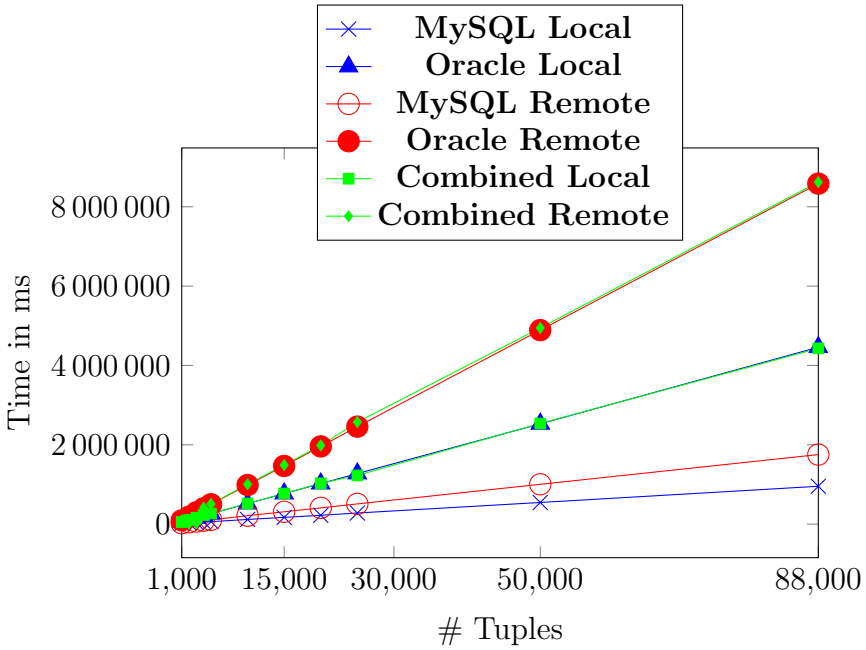



Figure 6.3: Initial *SeDiCo* Response Time

The figures from the *local* and from the remote evaluation in Figure 6.3 show that the average response time of the FVPD data is considerably slower compared to the initial implementation in Figure 6.2. The average response time for non-partitioned and non-distributed data compared to the average FVPD response time is ∼1,6 seconds for MySQL and ∼1,7 seconds for Oracle in the *local* setup and ∼2 seconds for MySQL and ∼3 seconds for Oracle in the *remote* setup. Compared to the FVPD setup in the initial *SeDiCo* approach, the response times are ∼257 seconds for a MySQL and ∼1,100 seconds for an Oracle database (*local*) and ∼465 seconds for MySQL and ∼2,200 seconds for Oracle in a *remote* setup.

### 6.3.1 Conclusion

Based on these figures, it has to be noted that the *SeDiCo* approach as is, unfortunately is not usable in practical usage scenarios. Above that, the response time for the combined measurement of MySQL and Oracle in Figure 6.3 shows interesting results, as both, the *local* and the *remote* values, are similar to the

results of the Oracle database. Therefore, the bottleneck in combined scenarios is always the slower database.

Comparing the *local* against the *remote* setup of this section shows that with a 1 Gbit Ethernet broadband connection between the components, the network causes another performance degrade by factor ∼2 averagely. This is caused by the client/server protocols of the used database systems and has the following reasons:

- Data being sent from a MySQL database through the network is encrypted via SSL (MySQL, 2016) and the same holds for Oracle databases (Oracle, 2016).
- Data that have to be transferred via network are split into packets, where MySQL uses 16 MB (MySQL, 2016) and Oracle uses 32 MB (Oracle, 2016) for each packet in its standard configuration, and this configuration was maintained throughout all evaluations in order to produce reproducible results.

This section concludes the pure *SeDiCo* framework performance evaluation and determines the *upper bound* for the response time of this work (cf. Section 1.4). Now, Section 6.4 evaluates the 3 join mechanisms, which are built to reduce the response times measured in this section such that they are in the same order of magnitude as the initial evaluation in Section 6.2.

## 6.4 Query Rewriting Evaluation

In this section, the response time of *query rewriting* with its 3 FVPD join algorithms (*nested-loops*, *hash* and *sorted-merge join*) is evaluated.

The following figures (Figure 6.4 - Figure 6.6) illustrate the respective join implementation for the FVPD partitions in the *local* as well as in the *remote* environment.

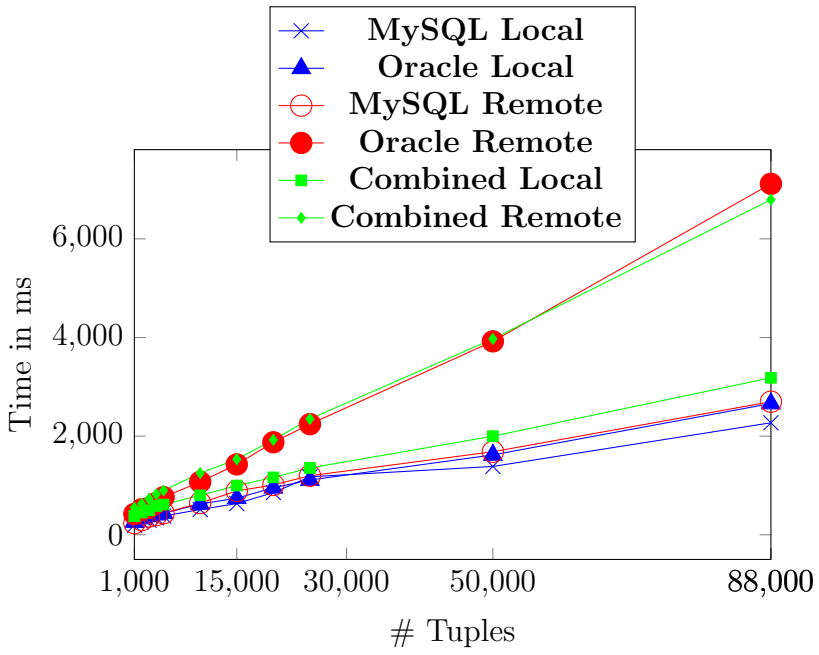



Figure 6.4: FVPD Query Rewriting Nested-Loops Response Time

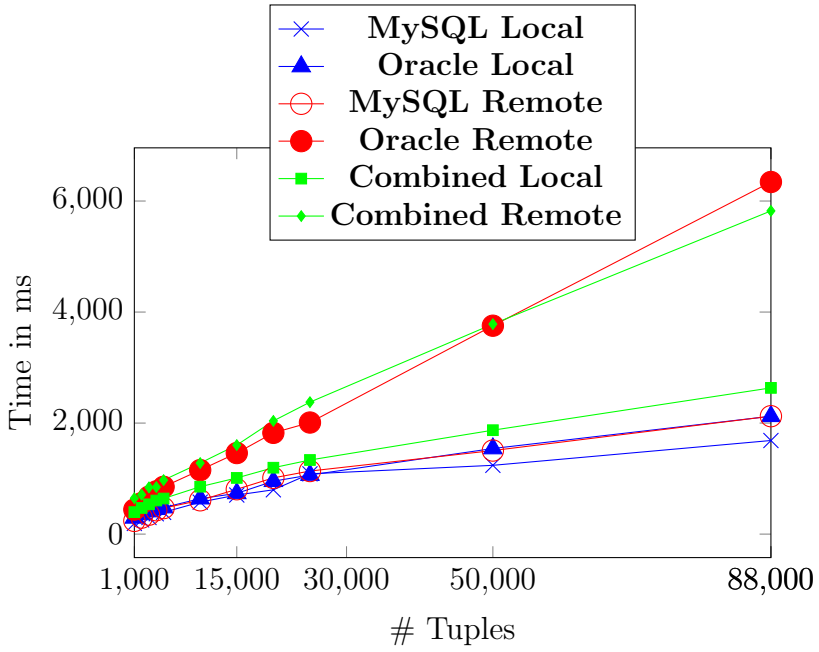



Figure 6.5: FVPD Query Rewriting Hash Join Response Time

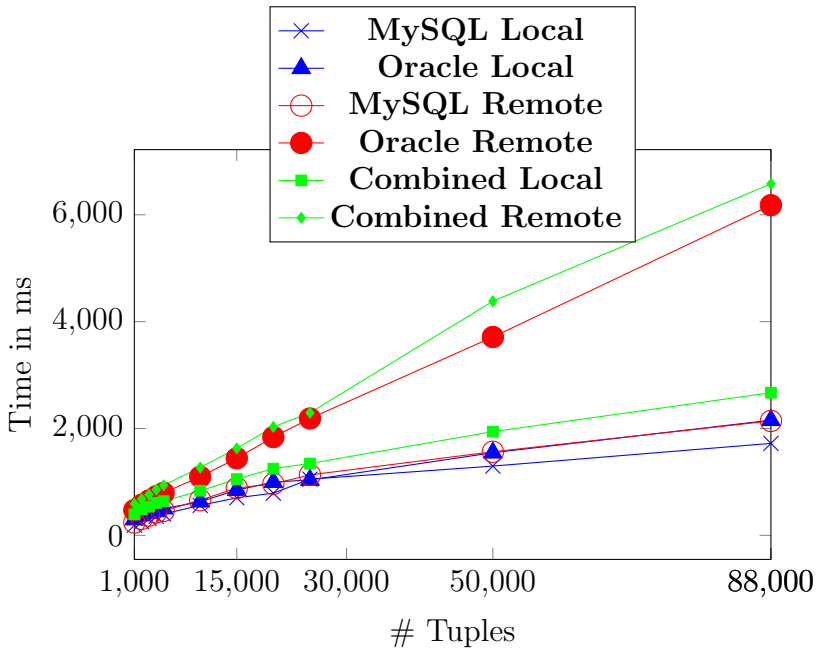



Figure 6.6: FVPD Query Rewriting Sorted-Merge Join Response Time

### 6.4.1 Conclusion

Similarly to (Kohler & Specht, 2015b), the *query rewriting* evaluation showed that the *hash join* and the *sorted-merge join* produced almost similar results. As expected, both outperformed the *nested-loops join* (Figure 6.4 - Figure 6.6) considering the average performance.

All join algorithms benefit from the fact that the rows are collected in sorted order based on their primary key values from the underlying FVPD database partitions. This reduces the join phase (join, probe or merge phase), as e.g. the inner loop of the *nested-loops join* can stop as soon as the first matching row is found. The same holds for the probe phase in the *hash join* and for the merge phase in the *sorted-merge join*.

Taking a closer look at the hash and the sorted-merge join, which both produced almost similar response times, it can be noted that collecting query matching rows from the FVPD partitions (i.e. the build and the sort phase) are similar. Both algorithms only differ in their join (i.e. their probe and merge) phases. The total response time of both algorithms depicted above show that the collection phase heavily predominates the join phase and that the join phase is an exceptionally small part of the total response time. Thus, even if the probe phase

(*hash join*) outperforms the merge phase (*sorted-merge join*) by factor ∼2, the overall response times of both algorithms are almost similar.

Regarding the *nested-loops join* performance in Figure 6.4, the results show a remarkable performance gain compared to the initial *SeDiCo* implementation depicted in Figure 6.3. Moreover, the *hash* and the *sorted-merge join* achieved even greater performance improvements as Figure 6.5 and Figure 6.6 show.

Another aspect with respect to the FVPD data, also mentioned in (Kohler & Specht, 2015b) is *skewness* of data. This phenomenon might emerge during the collection of the query matching rows from the FVPD partitions. Here, it might be the case that one partition contains a lot of query matching rows, whereas the other contains only few. Hence, heavily skewed data are advantageous especially for join algorithms. Figure 6.7[3] illustrates an evaluation of an extreme case in which one partition contains 288K and the other only 1 tuple. In this case the tuple fetching[4] phase dominates the overall performance, as only one tuple has to be joined.

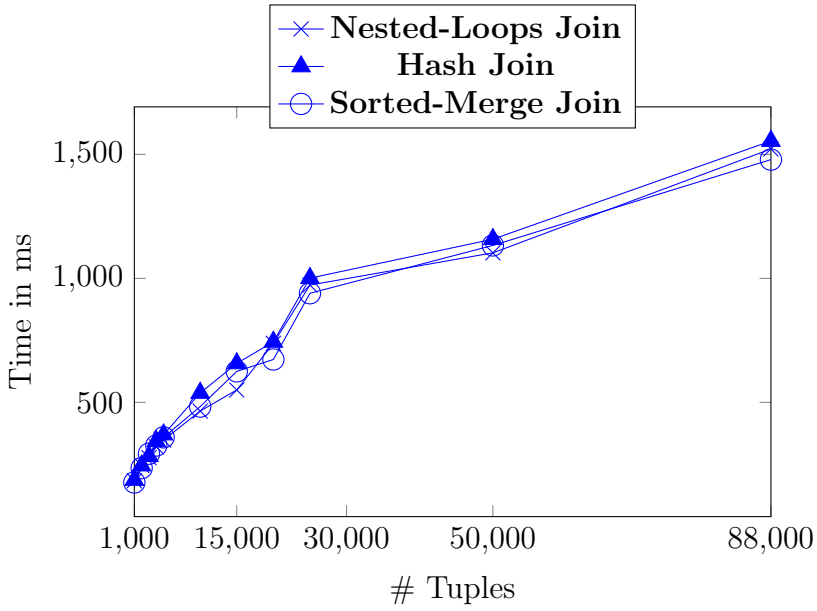



Figure 6.7: FVPD Query Rewriting Skewed Join Response Time

For this, the query was adapted such that it only returns 1 tuple for the first and 288K tuple for the second partition. The query issued against the FVPD MySQL data is as follows:

[3]For the sake of clarity, this evaluation was conducted in a local setup with a MySQL database.

[4]or build in *hash join*, or sort in *sorted-merge join*

Listing 6.1: FVPD Query for Skewed Data

```
1 SELECT * FROM Customer WHERE C_FNAME = 'gV;0{*:uxrXG^M' and
    C_YTD_PMT <=0;
```

Compared to these figures, Listing 6.1[5] shows faster collection times, as just one, particularly the larger partition with 288K rows has to be collected and not both. This explains the almost equal performance of all three join algorithms in the case of heavily skewed data. Above that, there is only one matching row that needs to be joined for the final result set. The results further show that the join phase is $1ms$ for all 3 algorithms. Surprisingly, the row collection phase is also equal in all 3 cases, which proves that the bottleneck is the database here. Further database measurements by the author proved that the overall performance in Figure 6.7 is exactly the time that is required to collect the 288K rows from one database partition.

This finally concludes the evaluation of *query rewriting* and now the next section deals with the performance measurement of *caching* with its centralized and decentralized implementations.

## 6.5 Caching Evaluation

In this section, *caching* with respect to its response time is evaluated.

This evaluation shows, how the performance differs in the *server-based* and in the *local* and *remote* caching approaches. As no modifications in the data set are performed, the pure cache performance is evaluated. This serves as a basis for further works that include cache synchronization, replication, invalidation and replacement strategies. As no tuple modifications occur during the evaluation, the respective caches can be filled with the entire data set which is known as *warming up* the cache and hence the cache contains the entire data set the underlying database system can be neglected. This also means that there are no cache misses and thus there is no distinction between MySQL and Oracle required in this section. In order to *warm* the cache, different approaches (e.g. SSD-based or the query rewriting) can be used. Therefore, this section only covers the pure cache

[5]This query further illustrates the randomly generated row values for the FVPD CUSTOMER table, described in Section 6.1.

performance and neglects the *cache warming phase*, as these values can be derived from the previous evaluation sections (Section 6.6 or Section 6.4).

This section starts with the evaluation of the *server-based caching* implementation, which is followed by the *local* and the *remote* one.

**Server-Based Caching** As in this implementation every partition has its own cache, none of these reside at the client. Thus, there is only a *server-based* evaluation for this implementation. Yet, this evaluation distinguishes between a *lazy* and a *parallel* row fetching strategy, which fetches rows from the caches either partition-wise or simultaneously in different threads. This is also outlined in more detail in Section 5.2.

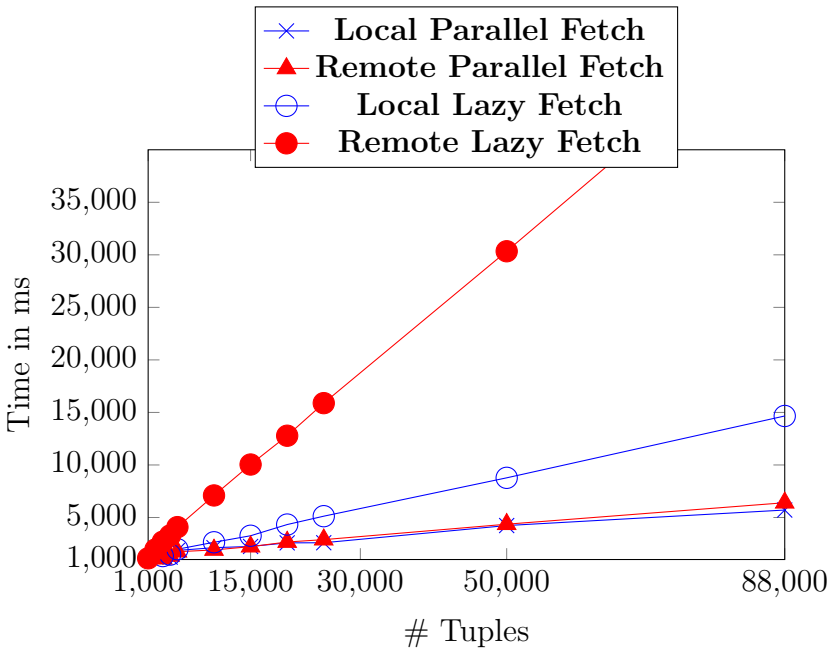



Figure 6.8: FVPD Server-Based Parallel and Local Response Time

**Local and Remote Caching** The results of the *local* and the *remote* caching approach are illustrated in Figure 6.9

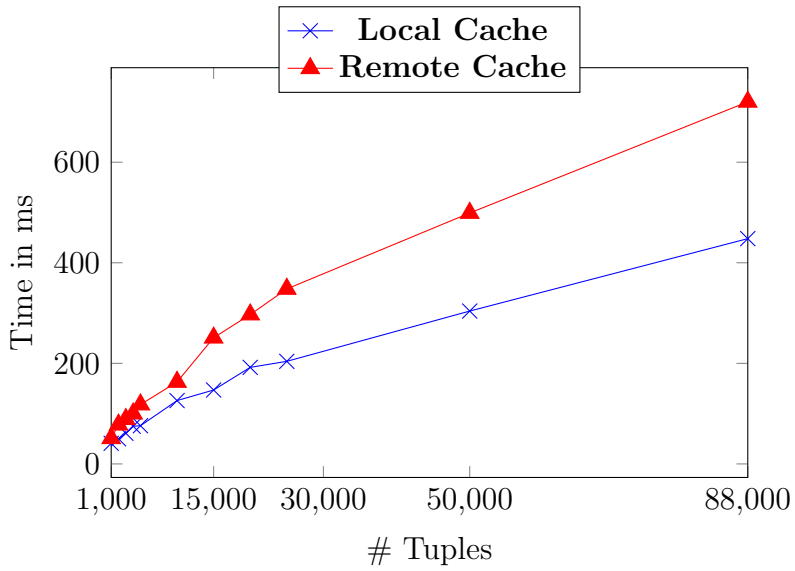



Figure 6.9: FVPD Local and Remote Caching Response Time

### 6.5.1 Conclusion

Considering the pure cache performance without the cache warming phase, both, the *local* and the *remote* implementations outperform the *server-based* one. This is not surprising, as in the *local* and *remote* caches, the rows are already joined and thus this is comparable to traditional non-partitioned and non-distributed database caching approaches.

This evaluation showed that the *local* and the *remote* implementations also outperform the *query rewriting* and the *SSD-based* approaches. However, taking the *cache warming* phase into consideration, the figures above get relativized and then *query rewriting* outperforms *caching* again.

The cache warming phase is neglected in this evaluation because it only has to be performed once, e.g. at the start of the *SeDiCo* client. Once the cache is warmed, all queries can be run against the cache (cf. Section 5.2. In addition to this however, updating the cache (e.g. regularly time-based, user-triggered, via cache invalidation, etc.) is another requirement, which is not considered in this evaluation. This is also too dependent on the specific database workload and regarded as a future work task in concrete application domains, where the *SeDiCo* framework will be integrated.

Moreover, considering the decentralized implementation it can be concluded that the parallel fetch outperforms the lazy fetch by factor ∼2 (local fetch) and

by factor ∼7 (remote fetch). However, the concrete fetch strategy is heavily dependent on the database workload and if the partitioning scheme is defined such that most queries can be answered with only the values of a single partition, the lazy fetch outperforms the parallel fetch, even if the partitions are accessed simultaneously there. To sum it up, the results confirmed the assumption that collecting rows form the cache memory is faster that directly from the FVPD partitions and this is the case for the entire caching approach.

## 6.6 SSD-based Evaluation

This section now covers the evaluation of the *SSD-based* implementation. As recent personal computer (PC) hardware and price developments show, SSDs are the most prominent and promising successors for traditional HDDs (i.e. secondary storage). In order to achieve performance improvements, the *SeDiCo* client and the database systems are operated on such SSDs and the response time of this setup is evaluated in this section in a *local* and in a *remote* environment.

**Basic SSD-based Performance Evaluation** Analogous to the previous evaluation sections in this work, firstly, a basic SSD performance metric with a non-distributed and non-partitioned data set is measured (Figure 6.10) and then the FVPD data set (Section 6.1) based on SSDs is evaluated in Figure 6.11.

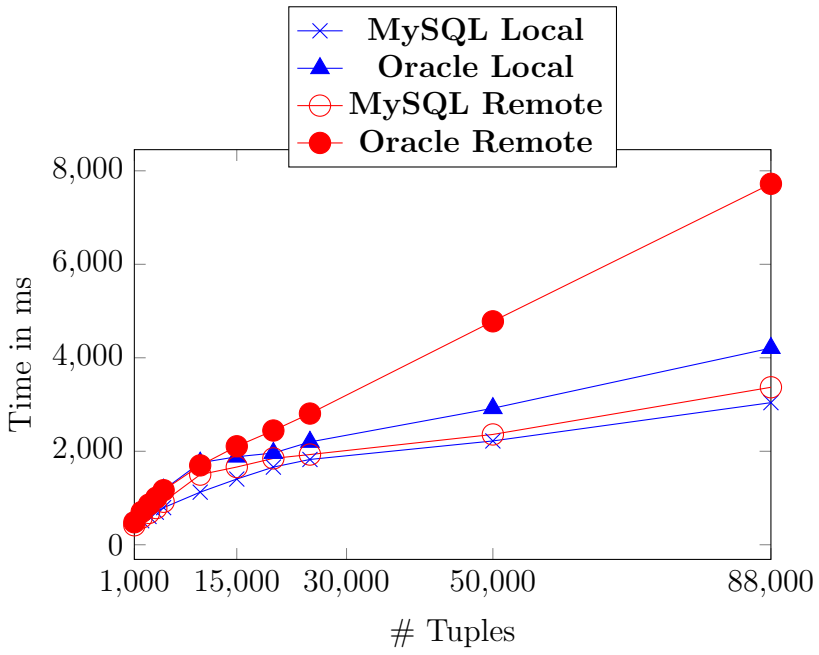



Figure 6.10: Initial Non-FVPD SSD-Based Response Time

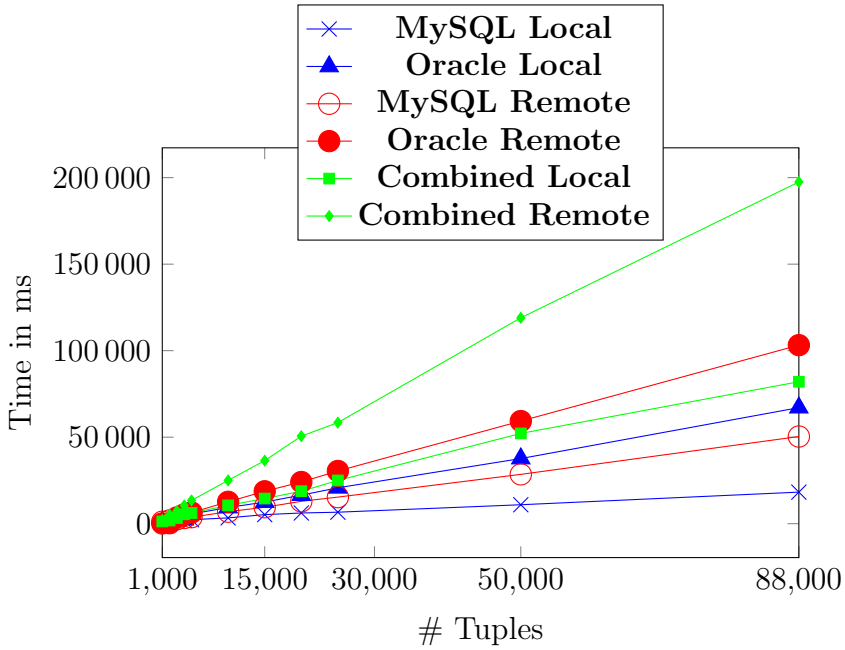



Figure 6.11: FVPD SSD-Based Response Time

### 6.6.1 Conclusion

The initial SSD performance measurement in the *local* and in the *remote* environment (Figure 6.10 compared to Figure 6.2), showed that indeed the SSD as secondary storage improves the response time by ∼30% in a *local* MySQL, by

∼20% in a *local* Oracle, by ∼60% in a *remote* MySQL, and by ∼32% in a *remote* Oracle environment (Figure 6.10). These are promising results; however, they show the improvements based on a non-partitioned and non-distributed data set.

Transferred to a FVPD data set, the performance gains are depicted in Figure 6.11 for the *local* and for the *remote* evaluation environment. Compared to the initial *SeDiCo* framework evaluation (Figure 6.3), these values show a significant performance gain by factor ∼50 (MySQL *local*), by factor ∼68 (Oracle *local*), by factor ∼34 (MySQL *remote*) and by factor ∼52 (Oracle *remote*).

Analogous to these figures are the measurements for the *combined* usage of MySQL and Oracle. Here again, the slower database system is the bottleneck, but the SSD-based approach improved the *combined response time* by factor ∼64 for the *local* and by factor ∼41 for the *remote* setup. To sum up this chapter, it can be concluded that although the usage of SSDs yields to a remarkable performance gain, the SSD-based approach is still not feasible in practical usage scenarios.

# Chapter 7

# Summarization of the Main Results

With the conclusions in Chapter 6, where every query strategy was compared against the initial *SeDiCo* implementation, this chapter now deals with the analysis of the query mechanisms and contrasts their evaluation results against the hypotheses formulated in Chapter 1. This chapter ends with a conclusion about all approaches and their strengths and weaknesses.

First of all, the hypotheses are either verified or rejected.

*Hypothesis 0: The definition of a FVPD Methodology ...*

This hypothesis can be verified. It was formally proved in Chapter 1. In addition to this, the *SeDiCo* framework as a concrete implementation of the FVPD methodology showed its technological feasibility.

With respect to the hypotheses 1-3, Table 7.1[1] states the aimed average response times of the non-partitioned and non-distributed data set, which was queried with Hibernate as the used ORM. The average response times used throughout this chapter are the average response times of the presented figures in the evaluation in Chapter 6, ranging from 1 - 288K tuples.

*Hypothesis 1: Query Rewriting ...*

This hypothesis can be verified. Surprisingly, the average performance of *query rewriting* and its corresponding join algorithms mostly outperforms the basic

[1] As these measurements are conducted on a non-FVPD data set, a *combined* measurement with both databases simultaneously was not possible.

Table 7.1: Average Hibernate Response Time for a Non-FVPD Data Set in ms

| Secondary Storage | MySQL Local | Oracle Local | MySQL Remote | Oracle Remote |
|---|---|---|---|---|
| **HDD** | 1,626 | 1,701 | 2,050 | 2,991 |
| **SSD** | 1,253 | 1,415 | 1,267 | 2,256 |

performance metric (Table 7.1). Accordingly, Table 7.2 shows the performance values of all query mechanisms with the best achieved values in bold font.

Table 7.2: Average FVPD Response Time in ms

| Approaches | MySQL Local | Oracle Local | MySQL Remote | Oracle Remote | Combined Local | Combined Remote |
|---|---|---|---|---|---|---|
| **Initial *SeDiCo*** | 257,346 | 1,177,436 | 465,972 | 2,266,895 | 1,178,949 | 2,287,508[a] |
| **Query Rewriting Nested-Loops Join** | **955** | 1,093 | 1,094 | 2,339 | 1,323 | 2,567 |
| **Query Rewriting Hash Join** | **600** | 755 | 716 | 1,944 | 979 | 1,959 |
| **Query Rewriting Sorted-Merge Join** | **609** | 766 | 729 | 1,946 | 993 | 2,255 |
| **Server-Based Caching Parallel Fetch** | **2,020** | N/A[b] | 2,094 | N/A[b] | N/A[b] | N/A[b] |
| **Server-Based Caching Lazy Fetch** | **4,186** | N/A[b] | 14,323 | N/A[b] | N/A[b] | N/A[b] |
| **Local Caching** | **146**[c] | N/A[b] | 365 | N/A[b] | 254 | N/A[b] |
| **Remote Caching** | **229** | N/A[b] | 434 | N/A[b] | 324 | N/A[b] |
| **SSD-Based** | **5,160** | 17,255 | 13,656 | 43,119 | 18,219 | 55,116 |

[a] note that this is the upper bound $t_{upper}$ defined in Section 1.3

[b] As the queries are directly issued against the cache, the underlying database can be neglected. Therefore, only MySQL was used for this evaluation.

[c] note that this is the lower bound $t_{lower}$ defined in Section 1.3, because here the local cache stores the already reconstructed relation $R(A)$

The results in Table 7.2 further show that the *hash* and the *sorted-merge join* have almost similar performance improvements. Yet, further evaluations with a larger data set (up to 1 million tuples) show that with more rows involved, the *sorted-merge join* outperforms the *hash join*, as the sort phase relies on the indexed primary key and is therefore faster than building the hash table[2]. Table 7.3[3] illustrates this in more detail.

Table 7.3: Comparison of Hash and Sorted-Merge Join with Larger Data Sets in ms

| **#Tuples** | **Hash Join** | | **Sorted-Merge Join** | |
|---|---|---|---|---|
| | **Build** | **Probe** | **Sort** | **Merge** |
| 1M | 30,207 | 6,097 | 339 | 3,895 |
| 750K | 10,167 | 1,671 | 578 | 3,859 |
| 500K | 6,491 | 1,357 | 375 | 3,831 |
| **Average** | **15,621** | **3,042** | **431** | **3,852** |
| **Total** | **18,063** | | **4,283** | |

The figures depicted in Table 7.2 and Table 7.3 show that *query rewriting* is absolutely applicable in practical usage scenarios. Hence, the entire *SeDiCo* framework becomes a viable approach with respect to security and privacy in especially public cloud environments.

*Hypothesis 2: Caching ...*

This hypothesis can be verified. For this evaluation, only the pure cache performance is important and thus Table 7.2 only focuses on the *local*, *remote* and *server-based* cache performance without the *cache warming* phase.

These values show that the pure cache performance of a *local* outperforms all other evaluated approaches. Although these values get relativized with the consideration of the cache warming, update and synchronization logic, these values are promising for the further *SeDiCo* framework development, where such cache coherence protocols will be developed.

Moreover, this evaluation showed that the *remote caching* approach is also viable in practical usage scenarios, if additional security and privacy preserving measures to secure the cache (e.g. VPN connections to and from the cache), that stores entirely reconstructed rows are taken into consideration.

[2]Basically, the sort phase is not required here, and the merge and the probe phases are equal.
[3]This short evaluation is based on a local MySQL installation.

The measurements in Table 7.2 clearly show that the average response time of the *server-based cache*[4] is also in the same order of magnitude as with a non-partitioned and non-distributed setup. However, this only holds for the *parallel fetch* strategy, where the FVPD partitions are queried in parallel. Especially, the remote *lazy fetch* in the evaluated setup was considerably slower (∼10) compared to the non-FVPD setup. Yet, it has to be mentioned (cf. Section 5.3, and (Kohler & Specht, 2015a)) that both partitions have to be queried, in order to reconstruct all tuples. In practical scenarios, it might be the case that only one partition must be queried and that tuples from the other one might not be required at all[5]. Then, the *lazy fetch* is even able to outperform the *parallel fetching* strategy.

Although the *server-based* caching implementation was outperformed by the *local*, by the *remote caching* and by the *query rewriting* approach (Table 7.2), it is applicable in practical usage scenarios. However, considering the evaluation results, it has to be mentioned that here also cache warming, update and invalidation mechanisms become crucial in scenarios where not the entire data set can be cached. Nevertheless, this implementation is a viable alternative with respect to the *remote* cache, concerning security and privacy aspects. As there exists one cache for every partition, the *Security-by-Distribution* principle is maintained and the caches can even be put into a public cloud infrastructure, which enables larger and dynamically scalable cache memories. In the end, this means that it is possible to cache entire data volumes in server-based caches, which is an interesting alternative for larger databases.

The results of the *SSD-based* approach are concluded accordingly and this refers to hypothesis 3.

*Hypothesis 3: Using Solid State Disks (SSDs) ...*

This hypothesis must be rejected. Although the response time gains achieved with SSDs were significant (cf. Section 6.6), the performance did not reach the same order of magnitude[6] as queries based on non-FVPD data sets. Nevertheless, the achieved performance values are listed in Table 7.2.

Although the average results are promising, the evaluation showed that especially for larger data volumes (i.e. $\geq 10K$ tuples) the response time loss in

[4]decentralized server-based cache
[5]if data is partitioned accordingly
[6]except for the local MySQL measurement

absolute values is not bearable in practical usage scenarios. This is the main reason for the rejection of this hypothesis.

Hence, with the confirmation of hypotheses 1 and 2 it can be concluded that *query rewriting* and the *caching* significantly contribute to the applicability of the *SeDiCo* framework in practical usage scenarios. Moreover, this work showed that every developed query mechanism has its raison dtre, as every approach realized remarkable performance gains compared against the original *SeDiCo* framework. The evaluation further showed that every query mechanism has pros and cons and therefore, no clear recommendation can be given here. Table 7.4 summarizes these advantages and disadvantages.

Considering the fact that the *build* (*hash join*) and the *sort phase* (*sorted-merge join*) are predominant in the total response time, parallelizing these steps would bring further performance improvements. Here, an implementation similar to the *parallel fetch* of the *decentralized server-based caching* cloud be possible.

Taking this thought one step further, the integration of the *query rewriting* approach into the *caching* approach could also be useful. As outlined in Table 7.4 a major drawback of *caching* is the (possibly long-lasting) cache warming phase. Hence, using *query rewriting* for warming the cache (and also for updating it), this disadvantage could be weakened. Accordingly, it would be possible to use faster cache memories with comparatively low cache coherence overhead.

This summary shows that *query rewriting* and *caching* proved the hypotheses of this work. Although the hypothesis concerning the *SSD-based* approach has to be rejected, even this approach promises performance improvements, however, the improvements were not as big as expected. Finally, it can be concluded that the results of *query rewriting* and *caching* are promising to further following *SeDiCo*'s vision of creating a *secure and distributed cloud data store* where the performance is in the same order of magnitude as traditional relational non-partitioned and non-distributed databases.

Table 7.4: Query Mechanism Summary

| Query Mechanism | Preserves *Security-by-Distribution* | Pros | Cons |
|---|---|---|---|
| **Query Rewriting** | yes | Fast response times | Large amount of client RAM memory necessary for the join algorithms (when large data volumes with many query matches are applied) |
| | | Applicable in practical usage scenarios | Advantages of parallel fetch can only be applied on clients that have multiple cores (i.e. as many cores as there are partitions) |
| **Caching** | | Fast response times | Additional cache coherence protocols, that affect the response time or the data consistency are required |
| | | Applicable in practical usage scenarios | |
| **Server-Based** | yes | Dynamically scalable cache memories | Slower response times compared to query rewriting and local and remote caching |
| **Local** | yes | Fastest response time compared against the other approaches | Cache warming required at every start of the client (the more data, the more time-consuming is the cache warming)<br>Cache requires large amount of client RAM (with large data volumes) |
| **Remote** | no | Cache warming must only be performed once at server start | Cache requires additional security and privacy measures |
| | | Dynamically scalable cache memories | |
| **SSD-Based** | yes | No conceptual, algorithmic or architectural *SeDiCo* framework changes required | Comparatively slow response times |
| | | | Inapplicable in practical usage scenarios because of the slow response times |

# Chapter 8

# Framework Application in Semantic Web Databases

This chapter describes a concrete application scenario for the FVPD approach in which it is transferred to Resource Description Framework (RDF)-based data. RDF is a standard for describing various kinds of resources in todays Semantic Web applications.

The introduction to the chapter outlines the history of the Semantic Web and its development. It further introduces basic notions and standards used in the Semantic Web and gives references to them in order to provide a basic knowledge about the used concepts. This is followed by the problem formulation in Section 8.2 that addresses data security and privacy in Semantic Web applications with a strong focus on RDF. Moreover, it formulates the hypothesis that the FVPD approach is applicable for both, a relational and a RDF-based data set at comparable response times. This is then followed by an outline of related works concerning distributed RDF-based data and the access to them via SPARQL Protocol and RDF Query Language (SPARQL) in Section 8.4. After that, the FVPD approach is conceptually transferred to RDF-based data in Section 8.5. Section 8.6 outlines the concrete implementation of the presented approach. This implementation is then evaluated in Section 8.7 and the evaluation results are discussed in the following Section 8.8. In the end, Section 8.9 covers a detailed view on further optimization strategies and challenges concerning federated RDF data stores based on FVPD relational data.

## 8.1 Introduction

In recent years there has been a development originating form the *World Wide Web* (WWW, or Web 1.0) where information was made available with standardized markup languages (XML, HTML, etc.) over standardized protocols (HTTP) towards a *Web of Participation* (Web 2.0) with social networks, blogs wikis, etc. and corresponding feedback and recommendation possibilities. Nowadays, broader network bandwidths and the easy access to information at any place and at any time (with e.g. mobile devices) lead to an information overload in which finding relevant information is becoming increasingly difficult and inefficient. The main research focus of the *Semantic Web* (Web 3.0) is to find approaches to structure information such that they can even automatically be processed and understood by machines. Actually, there are standards (or at least recommendations) from W3C for structuring information in the Semantic Web. At the moment, there is the Resource Description Framework (RDF) (Manola et al., 2014) based on XML that not only structures information as triples (subject, predicate, object) but also connects information with each other in form of links (predicates). Technologically, such triples are encoded as International Resource Identifiers (IRIs)[1]. Another standard relevant in this context is the SPARQL Protocol and RDF Query Language (SPARQL) (Harris & Seaborne, 2013) inspired by the SQL from relational databases which is one possible way[2] to access information in the Semantic Web.

Basically, RDF and SPARQL are examples that show that all Web 3.0 standards and their technological implementations rely on concepts (e.g. XML, HTTP, URIs, etc.) from the first version of the WWW (Web 1.0). SPARQL as the common query language to access RDF data is inspired by the well-known and exhaustively investigated SQL standard. Figure 8.1[3] illustrates this with an adaption of the Semantic Web Stack, and yet there are Ontology Based Data Access (OBDA) frameworks that create a mapping between relational data sources and their RDF-based representations and expose relational data as SPARQL endpoints.

Semantic Web frameworks (e.g. Virtuoso (Virtuoso, 2016), Apache Jena (Apache, 2016c), Sesame (Sesame, 2016), GraphDB (Ontotext, 2016), etc.) pro-

[1] IRIs are a superset of URIs which contain more Unicode characters. Further information can be found in (Duerst & Suignard, 2005)

[2] another possibility to access information is to browse through RDF data and follow the links between them with Semantic Web browsers, examples can be found at (W3C, 2016b)

[3] adapted from (W3C, 2007)

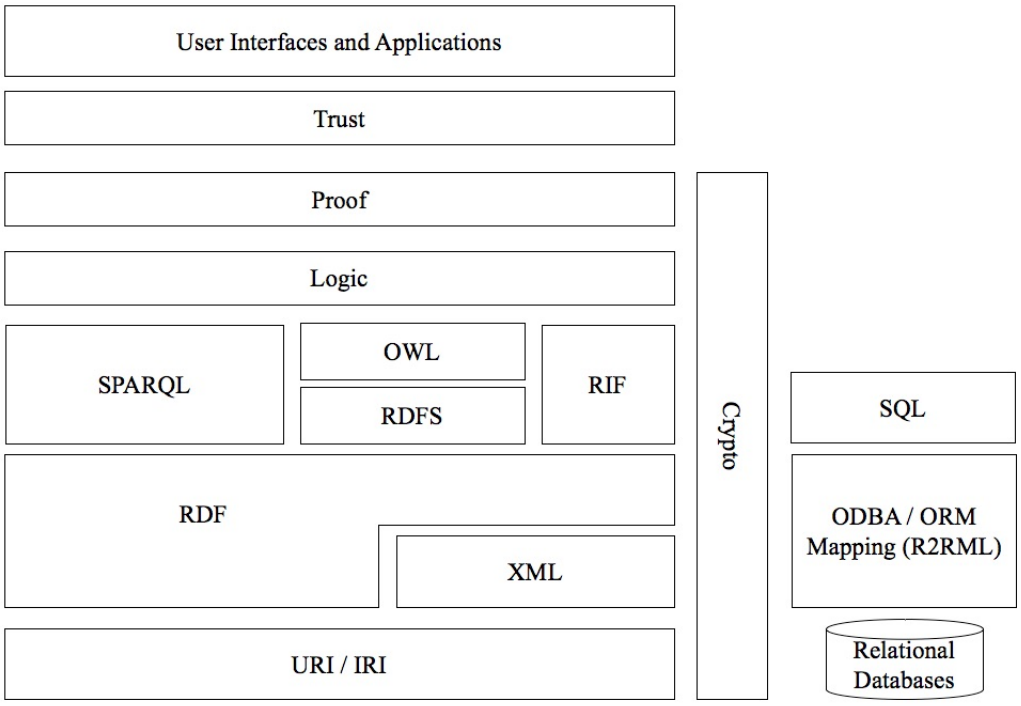



Figure 8.1: Adapted Semantic Web Stack

vide different storage engines for RDF data and corresponding SPARQL implementations to access them. Figure 8.2 gives a generic architectural overview about current storage engines and their interplay with the clients that issue SPARQL queries.

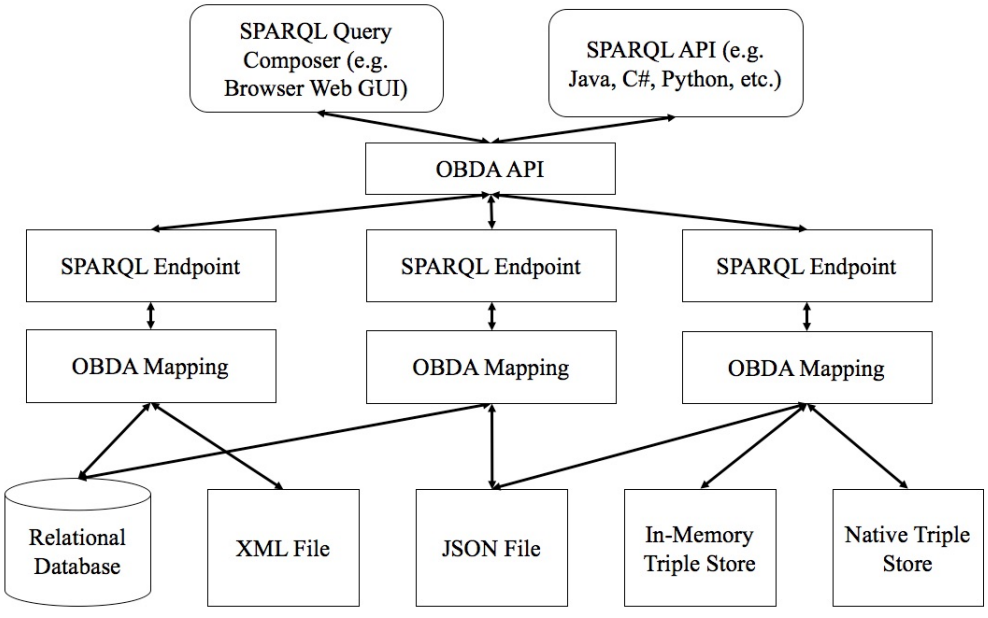



Figure 8.2: General Semantic Web Framework Architecture

The big advantage of describing and structuring heterogeneous data in a standardized way has been fostering the dissemination of RDF and corresponding data stores in recent years. Yet, with the usage of RDF, the underlying data stores have also experienced remarkable growth rates. This data growth can be summarized as Linked Data (LD) which not only benefits from the structured character of RDF but also from the possibility to connect data through links in form of RDF predicates. Moreover, Tim Berners-Lee published 5 star criteria that define how data should be published in the Semantic Web in order to be classified as Linked Open Data (LOD) (Berners-Lee, 2009). Again, these criteria show the usage of standard Web 1.0 technologies such as availability on the web, in machine-readable form (non-proprietary formats) using W3C standards like RDF and SPARQL and data should be connected with each other and set into context. A prime example of LOD is DBPedia (DBPedia, 2016) which connected ∼7 billion RDF triples (DBPedia, 2016) in April 2015. These statistics show that storing and processing such data in a single central repository is becoming more and more impractical. This is a challenging task that the W3C addressed to some extent with SPARQL version 1.1 that enables so-called federated queries over several repositories respective SPARQL endpoints (Harris & Seaborne, 2013). Besides these growing RDF data volumes, the possibility to provide data with SPARQL endpoints also fosters the dissemination of these endpoints. Hence, with a larger number of endpoints the demand to query them in a standardized way also increases and this was another reason for the support of federated SPARQL queries in the standard (Rakhmawati et al., 2013). Accordingly, as all data are stored as RDF triples, no additional layer to merge different resources, data types or formats (like ORMs in relational databases) are required (Görlitz & Staab, 2011).

These are all promising issues that draw the attention to federated RDF stores, however on the contrary it has to be noted that distributing data requires joining them again when they are accessed. This challenging task is similar to the join challenge in distributed databases outlined in Section 2.2 which decreases the data access performance significantly. Another issue that emerges with an increasing number of different SPARQL endpoints refers to the usage of different SPARQL versions. Here, it is a challenging task if an endpoint does not support SPARQL 1.1 but only its predecessor SPARQL 1.0. This restricts the language features, as e.g. the *SERVICE* keyword cannot be used then. Hence, SPARQL 1.1 queries issued against SPARQL 1.0 endpoints may not return any results (even if there were)

or a respective error message. Thus, it will be interesting how future standards handle this issue of backward compatibility.

Then, there is an ongoing discussion in the scientific community whether federated RDF stores and SPARQL endpoints are practical and viable approaches or not (Wu et al., 2014) (Görlitz & Staab, 2011) (Rakhmawati et al., 2013) (Haase et al., 2010) (Betz et al., 2012) (Betz, Hose, Sattler, 2012)[4].

Considering these aspects, the transformation of *SeDiCo*'s FVPD approach to new Semantic Web technologies is considered both, a viable approach to foster security and privacy-aware design considerations as well as an approach to use it as a concrete application scenario for further evaluation and dissemination tasks. Since the basic technological foundation (XML, HTTP, etc.) of the Semantic Web is well-investigated, only relevant aspects, namely RDF and SPARQL, for the adaption of the *SeDiCo* framework are outlined in more detail in the following sections.

Above that, as technologies related to the Semantic Web are a huge area of research, they cannot be extensively covered in this thesis. Hence, this chapter concentrates on RDF-based data that are based on relational data.

### 8.1.1 RDF

An exhaustive introduction about RDF is provided by the W3C in (Manola et al., 2014). Thus, this work only describes central aspects of RDF which are then later required for the implementation and the proofs of the approach. In RDF, all data are stored in form of triples denoted as subject, predicate and object; moreover, such triples can be visualized as a directed graph.

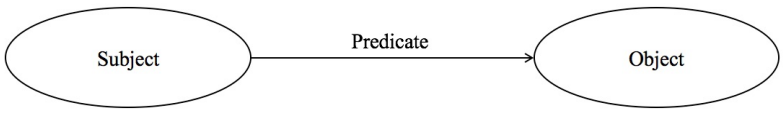



Figure 8.3: General RDF Triple

[4]For the interested reader, the mentioned literature here points to interesting advantages and disadvantages of single central and multiple federated repositories.

Listing 8.1: *CUSTOMER* RDF Triple Encoded in Turtle Syntax

```
1  ...
2  @prefix vocab: <http://kohlerjens.de:10001/CUSTOMER/resource/
       vocab/> .
3  ...
4
5  <http://kohlerjens.de:10001/CUSTOMER/resource/CUSTOMER/1>
6    a vocab:CUSTOMER ;
7    rdfs:isDefinedBy <http://kohlerjens.de:10001/CUSTOMER/data/
         CUSTOMER/1> ;
8
9    vocab:CUSTOMER_C_ID 1 ;
10   vocab:CUSTOMER_C_EMAIL "Cust@email.de" ;
11
12 ...
```

In this graph (or RDF statement), every node and the predicate are implemented as IRIs[5] that describe a resource which can be anything (e.g. real-world objects like documents, numbers, persons, etc.) (Manola et al., 2014). Transferred to the motivating *SeDiCo* example stated in the introduction of this thesis, such an RDF triple which states that a customer has a certain email address could be illustrated as follows.


Figure 8.4: RDF *CUSTOMER* Triple

Figure 8.4 shows that a *CUSTOMER* (*subject*) is identified by its unique IRI. Then the *predicate* states the relation between this *CUSTOMER* and its email address (denoted as the *object* which is also an IRI).

Moreover, RDF triples are technically storable in various serialization formats, e.g. JSON, XML, Turtle, etc. and most of the standard formats are derived from XML (Manola et al., 2014). Listing 8.1 illustrates the RDF triple from Figure 8.4 in Turtle syntax, shortened to the relevant aspects.

Listing 8.1 starts with the definition of the SPARQL endpoint as the *vocab* variable. Hence, it can be used in the entire RDF as an abbreviation for a better readability (line 9 and 10). Line 7 shows the reference to the schema (analogous

[5]The *subject* and the *object* can also be *blank nodes* or *literals* which is not relevant in this context, and therefore, more information can be found in (Manola et al., 2014).

to an XML schema) for the concrete definition[6] of the RDF. Furthermore, this RDF defines a single *CUSTOMER* with its id (*CUSTOMER_C_ID*) (line 9) and its email address (*CUSTOMER_C_EMAIL*) (line 10).

This triple representation leads to a great flexibility, as structured as well as unstructured data can be described in an uniform way (Duan et al., 2011) and so, huge RDF data silos like DBPedia (DBPedia, 2016) which collects and structures data from Wikipedia in RDF-based form have been implemented. Currently, there is a W3C recommendation for RDF version 1.1 (Manola et al., 2014) that extends RDF 1.0 with more data types, serialization formats and other details (Wood, 2014).

However, not only is it important to structure data and transform them in machine-readable and understandable form, it is also crucial to have a standardized query language to access RDF data in an uniform way. Thus, SPARQL was developed and also recommended (in version 1.1) by W3C (Wood, 2014).

### 8.1.2 SPARQL

In order to access RDF data via SPARQL Protocol and RDF Query Language, they have to be exposed as so-called SPARQL endpoints. SPARQL queries are then issued against these endpoints and the results (if there are) are returned. For the sake of brevity a detailed discussion about SPARQL and its language constructs the interested reader is guided to (Harris & Seaborne, 2013). This section only covers those language constructs necessary for implementing SQL equivalent SPARQL queries in the context of the FVPD CUSTOMER scenario.

A generic SPARQL query that retrieves all RDF data can be states as follows:

Listing 8.2: Generic SPARQL Query

```
1 @prefix pre: <http://kohlerjens.de/CUSTOMER/resource>
2 SELECT * WHERE { ?subject ?predicate ?object .}
```

This can be regarded as semantically equal to an SQL query that queries for all rows of a certain relation as depicted in Listing 8.3.

Listing 8.3: Generic SQL Query

```
1 SELECT * FROM <table_name>
```

[6] and e.g. for the validity check of the RDF

Listing 8.2 compared against Listing 8.3 illustrates fundamental differences as well as similarities of SPARQL and SQL:

- A SPARQL query is always formulated as a query for triples, so the WHERE clause must always contain a triple statement in form of subject, predicate and object.
- The semantic meaning of *WHERE* (Listing 8.2) and *FROM* (Listing 8.3) is equal. In SPARQL, the endpoint is usually defined as a prefix and queries are usually issued against a single endpoint that contains all relevant data. Indeed, the FROM keyword in SPARQL can be used to query different endpoints, however this is advanced in SPARQL 1.1 (Harris & Seaborne, 2013) where the *SERVICE* keyword is introduced.
- Both, SPARQL and SQL return unordered sets as result sets, containing sets of triples or sets of tuples respectively.
- Most of the SQL keywords (e.g. *ORDER BY*, *DISTINCT*, *LIMIT*, etc.) can also be used in SPARQL and have the same semantic meaning.
- The close relation between SPARQL and SQL is also elaborated in more detail in (Cyganiak & Cyganiak, 2005) where SPARQL is reduced to Codds relational model (Codd, 1970) which is also the foundation for SQL.

In order to not losing the scope of the chapter, further SPARQL language constructs (i.e. DESCRIBE, CONSTRUCT, ASK or further data manipulation constructs) are not discussed here. For this, the interested readers attention is drawn to (Harris & Seaborne, 2013).

The goal here is to transfer the FVPD approach to the architecture depicted in Figure 8.2. With respect to this, the FVPD *CUSTOMER* data set (cf. Section 6.1) is exposed as SPARQL endpoints (one endpoint for each partition) in order to provide a security and privacy-aware access to the data. Above that, the response time of this setup is compared against a non-partitioned and non-distributed *CUSTOMER* data set, also exposed as a single SPARQL endpoint. In order to remain comparable to the previous evaluation in Chapter 6, the same data set from the TPC-W benchmark and the same physical evaluation infrastructure is used. This chapter shows how the response time is affected when relational data are exposed as SPARQL endpoints in RDF-based form and how the FVPD approach influences the response time.

The integration of the FVPD approach in such a federated architecture has several implications and raises various research questions which are outlined in the following problem formulation in more detail.

## 8.2 Problem Formulation

The author of this work proposes to include security and privacy into to context of *linked data* (LD). Generally, as the name LD suggests, data should be linked. However, it might not be clear at first sight which data are confidential or sensitive or even worse, which data might become confidential and sensitive when they are combined with other data. Hence, the proposed FVPD approach to increase the level of security and privacy might also become viable in the context of LD. Furthermore, in relevant literature no approach has dealt with security and privacy in this context so far. Above that, it is worth mentioning that not even one of the W3C standards or recommendations considered security, privacy or performance in LD. Indeed, there are 2 papers (Rakhmawati et al., 2013) and (Betz et al., 2012) that mention copyright, data ownership and security, but only in a small section. Therefore, the challenge of privacy and security is considered as neglected so far. To substantiate the FVPD approach for LD, the following 4 best practices (Berners-Lee, 2006) are worth mentioning:

- IRIs (or URIs) to uniquely identify LD resources should be used
- HTTP as the basic protocol for the WWW (and therefore also for the Semantic Web) should be embedded in the IRIs
- LD resources should be published with standard protocols, i.e. RDF and SPARQL
- resources should be linked with each other.

This chapter introduces the FVPD approach for RDF-based data in order to improve the level of security and privacy and it measures how this approach affects the response times of the FVPD RDF-based data. This results in a comparison between the different SPARQL versions, namely version 1.0 and 1.1, and between a FVPD and a non-distributed and non-partitioned RDF data set, based on the TPC-W benchmark.

The architecture of the approach is illustrated in Fig. 8.5 and in Fig. 8.6. Here, relational data (i.e. CUSTOMER data) are exposed as SPARQL endpoints (via OBDA mappings), and thus, SPARQL query engines (e.g. Jena, Sesame, etc.) become able to query these relational data.

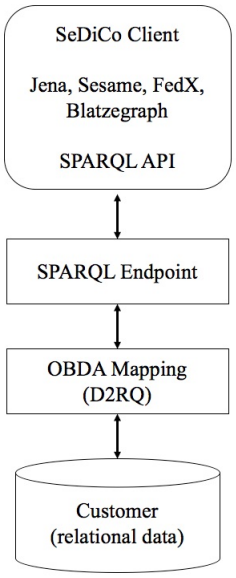



Figure 8.5: TPC-W *CUSTOMER* Table as SPARQL Endpoint

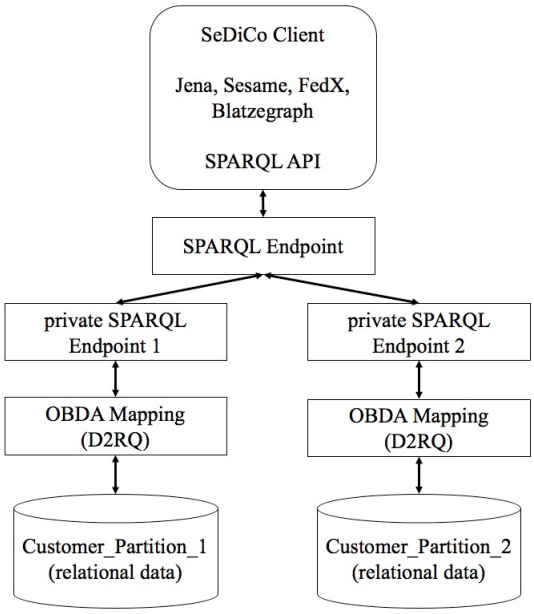



Figure 8.6: FVPD TPC-W *CUSTOMER* Partitions as SPARQL Endpoints

In contrast to federated RDF-based data stores, in the approach of this chapter, there is no knowledge required which data (i.e. triples) are stored in which RDF data store. In this approach, a SPARQL query is issued against a SPARQL endpoint and the framework determines (via SPARQL query rewriting) which private SPARQL endpoints are required to collect the queried triples. Thus, the actual storage location of the triples remains hidden, and authorized clients are able to query the endpoints without knowing their exact location or their addresses, etc. To the best of the author's knowledge this approach has not been followed so far.

From a security and privacy point of view, it must be considered that (besides the private SPARQL endpoints) the SPARQL endpoint for the original client queries should also not be publicly available; it should represent an internal endpoint, because of security and privacy-relevant data. On first sight, this is contrary to the LD principles, but adapting the LD principles to private data might be also a promising aspect.

Finally, this chapter is driven by a hypothesis that is stated as follows:

*Relational data, with respective mappings exposed as RDF data and published via SPARQL endpoints are vertically partitioned and distributed according to the FVPD methodology. Thus, the FVPD approach improves the level of security and privacy through physical and logical data distribution at comparable response times which are in the same order of magnitude as in a non-partitioned and non-distributed data set.*

## 8.3 Formal Definitions

Before the actual formalization, the correctness proof and the complexity analysis, a short remark concerning the *Open* and *Closed World Assumption* has to be given, to illustrate that the described RDF-based data distribution approach does not affect the semantic gap between *incomplete* data (*Open World Assumption*) and its standard query language SPARQL, which operates under the assumption that the underlying data is *complete* (*Closed World Assumption*).

### 8.3.1 Open and Closed World Assumption

This differentiation is outlined in greater detail in (Darari et al., 2014). In their work, they refer to a previous work (Darari et al., 2013) where they add meta data to their RDF-based data source that describe the data as either *complete* or *incomplete*. Thus, they are able to describe the retrieved triples as either *certain*, *complete*, or *possible* answers to the respective SPARQL queries. This is an interesting and promising approach to bridge the semantic gap between RDF and SPARQL. Moreover, this approach shows that the *Open* or *Closed World Assumption* depends on the underlying data source and as the FVPD approach focuses on the equality of the retrieved result sets (see above) and neither the query nor the underlying data set are changed, the semantic gap is left untouched here.

*Hence, it has to be stressed that the focus of this chapter relies on complete data sets (closed world assumption). SPARQL queries which refer to incomplete data sets (open world assumption) are considered as a future work task.*

### 8.3.2 Correctness

For the correctness proof, the following definitions are developed (analogous to Section 2.1.2 for the relational approach) and proved in the following sections. The definitions are based on the distinction between a *non-FVPD data set* and 2 corresponding *FVPD data sets*[7]. Therefore, the notion of the different kinds of data sets has to be defined first.

The following formal notations are based on the seminal work of (Pérez et al., 2006) and many subsequent works use these formalisms as well.

**Definition 5.** *Triple*

*A* ***triple*** $t$ *is defined in form* $t(subject, predicate, object) \in ((I \cup B) \times (I) \times (I \cup B \cup L))$*, with* $I$ *as International Resource Identifies (IRIs),* $B$ *as blank nodes, and* $L$ *as literals.* □

*IRIs*, represented as strings, are a superset of Uniform Resource Identifiers (URIs) and they contain more Unicode characters to *uniquely* identify a resource (e.g. a document, a web page, an image file, a video file, etc.) in the World Wide Web.

A *blank node* denotes a resource for which a concrete IRIs has not been established yet. Basically, it can be regarded as a placeholder for a IRI that has to be defined (in the near future).

Lastly, a *literal* is a string representation of a concrete value for a triple (e.g. a name, a date, a number, etc.). It is also possible to determine corresponding data types (strings, integers, dates, etc.) for the value that the *literal* denotes.

Note that according to (Manola et al., 2014) a subject can either be an IRI or a blank node. A predicate can only be an IRI and an object can either be an IRI, a blank node or a literal.

In the thesis, the focus is on RDF data sets which corresponds to relational database.

**Definition 6.** *RDF data set for a relational database (RDB).*

[7] the approach with more than 2 FVPD partitions works analogously

*Let $A$ be a set of attributes $A = \{a_1, ..., a_n\}$, for a relation $R(A)$ with a key attribute $a_1$. This attribute schema and the relation are converted into a RDF data set with the following representation*[8]*:*

- **Class**

  *An RDF schema class prefix:R is defined by the triple t(prefix:R, rdf:type, rdfs:Class)*[9]*. The class prefix:R is called* $\mathbf{PK_{Class}}$ *— primary key class.*

- **Properties**

  *For each attribute $a_i \in A$, $1 \leq i \leq n$, there is a property prefix:$a_i$ defined by the triple t(prefix:$a_i$, rdf:type, rdf:Property)*[10]*. Each property prefix:$a_i$ is therefore called attribute property.*

- **Row data set**

  *Each row $\{r_{k1}, r_{k2}, ..., r_{kn}\} \in R(A)$ is represented as sets of triples in the following way:*

  - *t(prefix:$r_{k1}$, rdf:type, prefix:R)*

    *With respect to Definition 5, the subject of this triple contains the primary key attribute value ($r_{i1}$). Hence, it is called a* $\mathbf{PK_{Instance}}$ *— the primary key instance for the row in $R(A)$.*

  - *$\{$t(prefix:$r_{k1}$, prefix:$a_i$, $r_{kj}$) $\mid$ $1 \leq j \leq n\}$, where the (row) value $r_{kj}$ is considered as a literal — represented as a string.*

  *The rest of the rows in relation $R$ are represented according to triples in the above-mentioned form.*

*The entire relation $R(A)$ is represented as a union of the triples for all sets defined above, denoted in the following way:*

$$RDF(R(A)) = \{\ t(prefix{:}R,\ rdf{:}type,\ rdfs{:}Class)\ \} \cup$$
$$\{\ t(prefix{:}a_i,\ rdf{:}type,\ rdf{:}Property) \mid a_i \in A,\ 1 \leq i \leq n\ \} \cup$$

[8]Note that in RDF it is allowed that IRIs are abbreviated with *prefix* definitions for the corresponding namespaces. In the following formalization the IRIs namespace for all the triples is simply denoted as *prefix*.

[9]The *rdf:type* property is usually used to state that a resource is an instance of a class. The prefix *rdf:* is a namespace for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. The prefix *rdfs:* is a namespace for <http://www.w3.org/2000/01/rdf-schema#>; *rdfs:Class* defines all RDF Schema classes.

[10]*rdf:Property* denotes the class of all RDF properties.

$$\{\ t(prefix{:}r_{k1},\ rdf{:}type,\ prefix{:}R) \mid \forall\{r_{k1}, r_{k2}, ..., r_{kn}\} \in R(A)\ \} \cup$$

$$\{\ t(prefix{:}r_{k1},\ prefix{:}a_i,\ r_{kj}) \mid \forall\{r_{k1}, r_{k2}, ..., r_{kn}\} \in R(A)\ and$$

$$1 \leq j \leq n\ \}$$

*Hence, RDF(R(A)) is called* ***RDB RDF data set*** *based on A and R(A). For the sake of better readability, it is called $\mathcal{D}$ in the following sections.* □

Table 8.1 summarizes the mapping accordingly.

| **Relational Model** | **RDF** |
|---|---|
| Table | Class |
| Attribute | Property |
| Attribute name | Object |
| Row value | Literal |

Table 8.1: Mapping between Relational Model and RDF

Each $\mathcal{D}$ can be divided into two conceptional sets corresponding to the conceptual schema of relation $R(A)$ which in Semantic Web is called **Ontology**; Firstly, an *instance set* containing the primary key instances and secondly, a *value set* containing triples for the properties from the conceptual set with concrete values for the corresponding rows.

The rest of the chapter deals with relational database data exposed as RDF data sets. Here, each row in $R(A)$ can be depicted as a graph like in Figure 8.7.

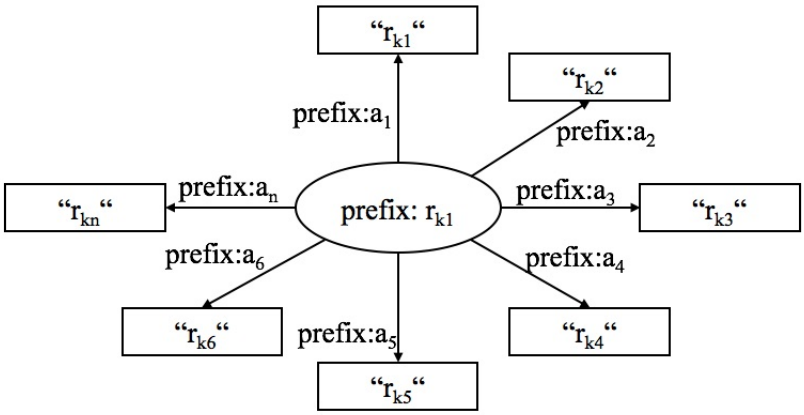



Figure 8.7: Graph for Primary Key Instance for a Row in $R(A)$

**Definition 7.** *SPARQL RDB Selection*

*Let A be a set of attributes $A = \{a_1, ..., a_n\}$, for a relation $R(A)$ with a key attribute $a_1$. Let $\mathcal{D}$ be the RDB RDF data set based on A and $R(A)$.*

*A SPARQL query of the form:*

*SELECT* $\{?object_i \mid for\ all\ i,\ 1 \leq i \leq n\}$

*WHERE* $\{\ \{?x\ \ prefix{:}a_i\ \ literal_i\ .\ \mid for\ some\ i,\ 1 \leq i \leq n\} \cup$

$\{?x\ \ prefix{:}a_i\ \ ?object_i\ .\ \mid for\ all\ i,\ 1 \leq i \leq n\}$

$\}$

*determines the set of all tuples that satisfy the conditions stated by the set of query patterns* $\{\ ?x\ \ prefix{:}a_i\ \ literal_i\ .\ \mid for\ some\ i,\ 1 \leq i \leq n\ \}$.[11]

*A SPARQL query of this form is called* **SPARQL RDB Selection query.**

*The result set of such a query is denoted by the following formula:*

$$RS \leftarrow [[\mathcal{Q}_{SELECT\ \{?object_i\ \mid\ for\ all\ i,\ 1 \leq i \leq n\}\ WHERE\ \{\ GraphPattern\}}]]_{\mathcal{D}}$$

*where*

$$GraphPattern = \{?x\ \ prefix{:}a_i\ \ literal_i\ .\ \mid for\ some\ i,\ 1 \leq i \leq n\} \cup$$
$$\{?x\ \ prefix{:}a_i\ \ ?object_i\ .\ \mid for\ all\ i,\ 1 \leq i \leq n\}.$$

□

Using a SPARQL RDB Selection query, tuples that correspond to $PK\_Instance$ having the $PK\_Instance$ as subject are selected. The statements in {?x prefix:$a_i$ literal$_i$ . | for some $i$, $1 \leq i \leq n$} impose restrictions on this $PK\_Instance$. Then, the statements in {?x prefix:$a_i$ ?object$_i$ . | for all $i$, $1 \leq i \leq n$} retrieve all values related to the $PK\_Instance$. The result set produced by this kind of query is similar to the result set produced by the relational database selection query (cf. Section 1).

**Definition 8.** *SPARQL RDB Projection*

*Let $A$ be a set of attributes $A = \{a_1, ..., a_n\}$, for a relation $R(A)$ with a key attribute $a_1$. Let $\mathcal{D}$ be the RDB RDF data set based on $A$ and $R(A)$.*

*A SPARQL query of the form:*

*SELECT* $\{?object_i \mid for\ some\ i,\ 1 \leq i \leq n\}$

[11] More exact the result is a set of tuples, where each tuple is a singleton containing a primary key instance.

*WHERE* { { *?x prefix:*$a_i$ *?object*$_i$ . | *for some* $i$, $1 \leq i \leq n$}

}

*determines a set of tuples formed by each of the variable in the list of variables in the query:* { *?object*$_i$ | *for some* $i$, $1 \leq i \leq n$}.

*A SPARQL query of this form is called* **SPARQL RDB Projection query.**

*The result set of such a query is denoted by the following formula:*

$$RS \leftarrow [[\mathcal{Q}_{SELECT\ \{?object_i\ |\ for\ some\ i,\ 1 \leq i \leq n\}\ WHERE\ \{\ GraphPattern\ \}}]]_{\mathcal{D}}$$

*where*

*GraphPattern* = { *?x prefix:*$a_i$ *?object*$_i$ . | *for some* $i$, $1 \leq i \leq n$}. □

Such a SPARQL RDB Projection query only retrieves a set of tuples. The values in the retrieved tuples correspond to the attribute properties used in the *GraphPattern*. A SPARQL RDB Projection query is denoted in the following way:

$$\Pi^{SPARQL}_{(a_i,\ldots,a_j)}\mathcal{D}$$

where $1 \leq i < j \leq n$. Furthermore, the abbreviated form $\Pi^{SPARQL}_{\omega}\mathcal{D}$, where $\omega = (a_i, \ldots, a_j), 1 \leq i < j \leq n$ is used for the sake of better readability. The result set is defined as:

$$RS \leftarrow \Pi^{SPARQL}_{\omega}\mathcal{D}.$$

Definitions 7 and 8 show that the basic difference between *SPARQL RDB Selections* and *SPARQL RDB Projections* are the way the filter conditions (i.e. literals or objects) are formulated. In both cases the result set is a set of tuples. Hence, the FPVD approach is applicable for both kinds of queries, but for the sake of better readability, the rest of the chapter is based on SPARQL RDB Projections. SPARQL RDB Selections work analogously. When it is clear from the context, *SPARQL Projection* or *Projection* will be used instead of *SPARQL RDB Projection*.

**Definition 9.** *FVPD and non-FVPD RDF data sets*

*Let $A$ be a set of attributes $A = \{a_1, ..., a_n\}$, for a relation $R(A)$ with a key attribute $a_1$. Let $\mathcal{D}$ be the RDB RDF data set based on $A$ and $R(A)$, and let two RDB RDF data sets $\mathcal{D}_{v1}$ and $\mathcal{D}_{v2}$ be defined, such that*

- $\mathcal{D} = \mathcal{D}_{v1} \cup \mathcal{D}_{v2}$

  *indicating that the union (i.e. the join) of the triples in $\mathcal{D}_{v1}$ and $\mathcal{D}_{v2}$ result in the original non-FVPD data set $\mathcal{D}$,*

  *and*

- $\mathcal{D}_{v1} \cap \mathcal{D}_{v2}$ =

  $\{$ *t(prefix:R,rdf:type,rdfs:Class)* $\} \cup$

  $\{$ *t(prefix:*$a_1$*, rdf:type, rdf:Property)* $\mid a_1 \in A \} \cup$

  $\{$ *t(prefix:*$r_{k1}$*, rdf:type, prefix:R)* $\mid \forall\{r_{k1}, r_{k2}, ..., r_{kn}\} \in R(A) \} \cup$

  $\{$ *t(prefix:*$r_{k1}$*, prefix:*$a_1$*,* $r_{k1}$*)* $\mid \forall\{r_{k1}, r_{k2}, ..., r_{kn}\} \in R(A)\}$

*Then, $\mathcal{D}_{v1}$ and $\mathcal{D}_{v2}$ are called **FVPD data sets** for the **non-FVPD data set** $\mathcal{D}$.* □

It is easy to show that $\mathcal{D}_{v1}$ and $\mathcal{D}_{v2}$ are the RDB RDF data sets based on lists of attributes $B$ and $C$ such that $A = B \cup C$ and two relations $S_{v1}(B)$ and $T_{v2}(C)$ according to Definition 6. Above that, the definition for more than two FVPD data sets works analogously and is not given here.

**Definition 10.** *Reconstruction queries*

*Let $A$ be a set of attributes $A = \{a_1, ..., a_n\}$, for a relation $R(A)$ with a key attribute $a_1$. Let $\mathcal{D}$ be the RDB RDF data set based on $A$ and $R(A)$ — a non-FVPD data set. Let $\mathcal{D}_{v1}$ and $\mathcal{D}_{v2}$ be two FVPD data sets for $\mathcal{D}$.*

*Let $\Pi^{SPARQL}_{(a_i,...,a_j)}, (1 \leq i < j \leq n)$ be a projection query for $\mathcal{D}$ such that*

$$RS \leftarrow \Pi^{SPARQL}_{(a_i,...,a_j)}\mathcal{D}.$$

*Let $\Pi^{SPARQL}_{v_1(a_1,a_k,...,a_l)}$ be a projection query for $\mathcal{D}_{v1}$ and let $\Pi^{SPARQL}_{v_2(a_1,a_m,...,a_o)}$ be a projection query for $\mathcal{D}_{v2}$ with $1 \leq i \leq k, l, m, o \leq j \leq n$, such that*

$$RS_{v1} \leftarrow \Pi^{SPARQL}_{v_1(a_1,a_k,...,a_l)}\mathcal{D}_{v1} \text{ and } RS_{v2} \leftarrow \Pi^{SPARQL}_{v_2(a_1,a_m,...,a_o)}\mathcal{D}_{v1}.$$

*The projections queries* $\Pi^{SPARQL}_{v_1(a_1,a_k,...,a_l)}$ *and* $\Pi^{SPARQL}_{v_2(a_1,a_m,...,a_o)}$ *are called* **reconstruction queries** *for the projection query* $\Pi^{SPARQL}_{(a_1,...,a_j)}$*, if and only if*

$$RS = \Pi_{(a_i,...,a_j)}(RS_{v1} \bowtie_{a_1} RS_{v2}).$$ □

Note that in the last equation the projection is over a table which represents the result from the join operation. Similarly to the relational database case (cf. Section 1) this last projection is necessary if the the property for the key attribute is not part of the conditions for the SPARQL Projection query.

Regarding all definitions above, it can be stated that the FVPD methodology (see Definition 4) is applicable in this restricted case where RDF data have a corresponding representation of relational data: in a sense that the representation of an RDB RDF dataset in several partitions does not contain any security or privacy relevant data.

The correctness of the approach is stated in Theorem 2.

**Theorem 2.** *Let* $A$ *be a set of attributes* $A = \{a_1, ..., a_n\}$*, for a relation* $R(A)$ *with a key attribute* $a_1$*. Let* $\mathcal{D}$ *be the RDB RDF data set based on* $A$ *and* $R(A)$ *— a non-FVPD data set. Let* $\mathcal{D}_{v1}$ *and* $\mathcal{D}_{v2}$ *be two FVPD data sets for* $\mathcal{D}$*.*

*For each* $\Pi^{SPARQL}_{(a_i,...,a_j)}, (1 \leq i < j \leq n)$ *projection query for* $\mathcal{D}$*, such that*

$RS \leftarrow \Pi^{SPARQL}_{(a_i,...,a_j)}\mathcal{D}$*,*

*there exist two projection queries* $\Pi^{SPARQL}_{v_1(a_1,a_k,...,a_l)}$ *for* $\mathcal{D}_{v1}$ *and* $\Pi^{SPARQL}_{v_2(a_1,a_m,...,a_o)}$ *for* $\mathcal{D}_{v2}$*, that are reconstruction queries for the original projection query* $\Pi^{SPARQL}_{(a_i,...,a_j)}$*.* □

The formulation of the algorithm for the creation of the two reconstruction queries and the proof of the theorem are a straightforward application of the approach taken for Theorem 1 given in Section 1.

### 8.3.3 Complexity

The *reconstruction* queries only iterate through the triples of each FVPD data set and if there are matches with other partitions, the triples are reconstructed, i.e. *joined*. As the *reconstruction* queries only have to iterate through every FVPD data set, this results in an overall complexity of $n^m$, with $n$ as the number of triples and $m$ as the number of partitions. Furthermore, as $m$ as the number of

partitions typically is rather small[12], the approach seems applicable in practical use case scenarios[13].

## 8.4 Related Work

The problem of querying federated SPARQL endpoints has emerged with a constant data growth (Quilitz & Leser, 2008). With respect to this, (Haase et al., 2010) developed requirements for an efficient way of querying several RDF repositories at once, which can be summarized as follows:

- The selection of the data repositories:
  - Which repositories are able to answer the query and which ones can be neglected?
  - How are the results handled if big repositories return a lot of results?
  - How often is data in a repository changed and how does that affect the query results?
- The lifecycle of a federation:
  - How long is a certain federation able to provide reliable, current, etc. results?
- The federated queries:
  - Are they short (easy) or long (complex) running queries?
  - Is it allowed to manipulate data in repositories?
  - What happens if a repository provider permits manipulations and another one forbids them?
- The endpoint definition:
  - How is the problem of different SPARQL and RDF versions dealt with?
- The Service Level Agreements (SLAs):
  - How are SLAs between client and provider negotiated?

[12] in the outlined approach of this chapter: m = 2

[13] although further evaluation with real world use cases have to be conducted as future work tasks

– What kind of SLAs are available and how are different SLAs managed?

These considerations show the challenging character of federated queries over several endpoints. RDF uses serializable formats for storing these heterogeneous data. This is contrary to the relational character of traditional databases and as most of current applications use relational databases as their foundation, it is considered unlikely that these relational data are all transferred to RDF-based formats. Hence, there are different approaches, called Ontology Based Data Access (OBDA) frameworks to expose relational data as SPARQL endpoints. A current W3C recommendation defines Relational Database to Resource Description Framework Mapping Language (R2RML) (Das et al., 2012) as a mapping language for relational data to RDF. This recommendation defines a mapping that transfers relations to RDF graphs and implementing Semantic Web frameworks are then able to expose these data as SPARQL endpoints. Hence, exposing relations as SPARQL endpoints can be implemented in basically two ways (Rodríguez-Muro & Rezk, 2015):

- On the one hand, relational data are mapped to RDF and these RDF triples are then stored in a native tuple store. Thus, the relational data remain untouched, however, loading and serializing relational data into RDF may require long lasting loading phases and the challenge of data consistency between the relations and the triples is complex to solve.
- On the other hand, relational data mapped to RDF and the OBDA framework translates the SPARQL queries according to the mapping into traditional SQL queries. This approach is also known as building a virtual RDF graph. It avoids the previously mentioned disadvantages at cost of performance and at cost of a complex query rewriting procedure from SPARQL to SQL.

These two approaches demonstrate the challenging question of implementing a single RDF repository (first approach) or dealing with several federated ones (second approach). As the introduction of the FVPD approach contradicts the single repository approach, the federated one is addressed in the remainder of this chapter. However, although the SPARQL 1.1 standard includes federated queries, their optimizations are still open research challenges and this is also the case for efficient transformations from SPARQL to SQL queries. In their empirical

study (Arias et al., 2011) the authors determined that in SPARQL queries the join operator is mostly used, besides the *SELECT* operator which is used in 99% of all queries[14]. Therefore, optimizations for the join of federated RDF data (similar to the join in the FVPD approach, cf. Section 2.2) still promises performance improvements. With respect to this, (Görlitz & Staab, 2011) give an exhaustive summary of possible join algorithms (e.g. remote join[15], mediator join[16] and further improvements of them) for queries over federated RDF repositories.

Summarizing these approaches results in attempts to find optimized query execution plans to improve the overall response time, i.e. the response time which includes the collection of the result sets, their join and their transport to the client. On the contrary, (Schmidt et al., 2011) claim that optimized query execution plans might be of limited usage especially in federated RDF data stores. They pointed out that such strategies mainly rely on the respective SPARQL endpoints meta data (repository size, response time or other SLAs) and that it cannot be predicted if a certain repository is suitable to answer a query or not. This is different compared to relational database scenarios and especially when the FVPD approach is applied where it is clear which relations are containing the respective results and where it is known which partitions are required to answer the query.

Despite the fact that a central repository is faster compared to a federated one, various Semantic Web frameworks support both approaches. However, it has to be noted that there is a trend to remove the support for exposing relational data as SPARQL endpoints because of the severe performance issues (Apache, 2015) (Sesame, 2015). This contradicts the W3C R2RML standard as well as the challenging fact that data volumes steadily increase. In contrast to this, there are approaches that further pursue the idea of federated data stores, like (Wu et al., 2014) (Harris et al., 2009) (Harth et al., 2007) where data are partitioned across several computing nodes aiming at answering SPARQL queries more efficiently (e.g. in parallel), as only smaller result sets have to be joined. Interestingly, (Wu et al., 2014) identified In-Memory hash join as the most commonly used join algorithm in federated RDF scenarios like applied in Section 5.1 for the relational FVPD data set.

[14]besides e.g. *ASK*, *CONSTRUCT* or *DESCRIBE* operators

[15]joins are processed instantly on the respective remote repository (if possible) and only the result sets are transferred to the client

[16]a central instance (mediator) collects all result sets and joins them in a central location

### 8.4.1 Caching

Other approaches concerning federated SPARQL queries use query caching. Here, (Martin et al., 2010) present a caching architecture analogous to traditional relational database caches (also depicted in Section 5.2) that keeps RDF triples in the cache as long as the subject, predicate or object is not manipulated. They were able to improve the response time by factor 10 with repositories that contain more than 1 million triples. However, this approach is more complex compared to relational database caching (i.e. relation or subset relation caching), as SPARQL always queries for concrete triples and if a query changes, the results cannot be collected from the faster cache memory, as they have not been cached yet (Martin et al., 2010). Hence, query caching is only considered applicable in scenarios where queries do not change very often.

### 8.4.2 Benchmarking

Related works concentrate on benchmarking the response time of single or federated RDF repositories. In recent years, there has been a discussion about the usage of artificial or real-world SPARQL queries (Morsey et al., 2011) (Qiao & Özsoyo, 2015) which influenced the benchmark development considerably. With respect to this, the following list mentions benchmarks (without being exhaustive) which are widely adopted and used throughout academic research as well as in industry projects:

- Berlin SPARQL benchmark (BSBM) (Bizer & Schultz, 2001)

  BSBM uses an e-commerce scenario in which customers post reviews about products offered by various vendors. This is an interesting benchmark as it is also considered viable to measure the response time based on relational data (Morsey et al., 2011). However, the data scheme shows that the customer scheme only contains three attributes (name, country, mbox_sha1sum) and this is not comparable to the previously used TPC-W benchmark in Section 6.1 of this work.

- DBPedia SPARQL benchmark (DBPSB) (Morsey et al., 2011)

  DBPSB focuses on unstructured Wikipedia data that is stored in RDF triples. Unfortunately, it lacks the structured character of relational data

exposed as SPARQL endpoints. Moreover, no further similarities with the TPC-W benchmark could be found.

- Leigh University Benchmark (LUBM) (Guo et al., 2005)

  LUBM uses an university scenario with students and professors, and no further similarities with the TPC-W benchmark scenario and its customer relation could be found.

- SP2 Benchmark (SP2) (Schmidt et al., 2009)

  SP2 is based on the Digital Bibliography Library Project (DBLP) (DBLP, 2016) which manages authors and their publications at conferences, in journals, etc. Hence, here also no similarities with the TPC-W benchmark could be determined.

- RBench (Qiao & Özsoyo, 2015)

  RBench also uses the DBPedia data set, but in contrast to SP2 it uses real-world queries instead of artificially generated ones which the previously mentioned benchmarks do. Hence, here also no parallels in the data set to the TPC-W benchmark could be found.

- FedBench (Schmidt et al., 2011)

  FedBench was defined for the usage in federated environments to measure the respective response time. It also uses various data sets from DBPedia, GeoNames (GeoNames, 2016), KEGG (Kanehisa Laboratories, 2016), etc. Again the lacking similarities with the TPC-W data set are the cause for not considering this benchmark any further in the context of this work.

These are reasons, beside the main reason which is the comparability with the previous chapters of this work, why the TPC-W benchmark was transferred to an RDF scenario where the *CUSTOMER* relation was exposed as one (single repository implementation) or more (federated repository scenario) SPARQL endpoints. Accordingly, it will be interesting to compare the results of the evaluation outlined in Chapter 6 with the response times of this chapter.

## 8.5 Approach

The general approach to expose relational data as RDF triples according to R2RML (Das et al., 2012), offers two possibilites. Firstly, data not only can be exposed as

RDF triples (and stored internally as relational data), but secondly, also converted into RDF triples. This conversion has the advantage that it abstracts from the underlying triple store and various tiple store (e.g. GraphDB (Ontotext, 2016), etc.) implementations can be used to store these triples. However, the approach in this chapter uses the first approach that exposes relational data as SPARQL endpoints in order to remain comparable to the previous chapters, especially to the evaluation in Chapter 6.

This approach benefits from the long history of relational databases and their optimizations, and this section contributes to the idea of transferring these concepts and exploiting them with RDF-based data. This leads to the hypothesis that traditional optimization techniques can be applied to RDF stores in order to benefit from performance improvements. This hypothesis was initially elaborated by (Sequeda & Miranker, 2013), but the authors were not able to clearly confirm it (their results only provide hints to support it). Moreover, they regarded commercial database systems with advanced optimization capabilities compared to Open Source implementations. They also stated that the optimizer is not always able to choose the best query optimization strategy (e.g. cost-based, rule-based, etc.). They concluded that a lot of optimization potential could be achieved by improving federated join operations, which have already been elaborated by mentioned works in Section 8.4.

Since there are Semantic Web frameworks that support exposing relational data as SPARQL endpoints, it is an interesting question how these frameworks perform when the FVPD approach is applied to the underlying relational databases. For this, the SPARQL query engines listed in (W3C, 2016a) were taken into consideration and analyzed for their suitability based on the criteria mentioned in Table 8.2. Here for the sake of clarity, only the most prominent query engines with a strong focus on their current development states and their maintainability are analyzed any further.

- FedX (FluidOperations, 2016)

  FedX, an Open Source implementation, is able to query federated SPARQL endpoints. It also provides full SPARQL 1.1 support in its latest version. On the one hand, it has to be mentioned that FedX is based on the Sesame framework for accessing various kinds of RDF data stores. On the other hand, FedX abstracts from Sesame and integrates federated query capabilities (e.g.

Table 8.2: Semantic Web Frameworks Analysis

| Criteria / Query Engine | Currently Developed and Maintained | SPARQL 1.1 Support | Open Source Implementation Available | SPARQL Endpoint Access | Support for Relational Data Stores |
|---|---|---|---|---|---|
| **FedX** | Yes | Yes | Yes | Sesame | Yes |
| **Apache Jena** | Yes | Yes | Yes | Jena | Deprecated |
| **Sesame** | Yes | Yes | Yes | Sesame | Yes |
| **D2RQ** | Yes | No | Yes | Jena, Sesame, and others | Yes |
| **Virtuoso** | Yes | Yes | Limited version | Jena, Sesame and others[a] | Yes |
| **Ontop** | Yes | Preliminary | Yes | Sesame[a] | Yes |
| **Blazegraph** | Yes | Preliminary | Yes | Sesame[a] | No |
| **GraphDB** | Yes | Yes | Limited version | Sesame, Jena[a] | No |
| **DARQ** | No[a] | No | Yes | Jena | Yes |

[a]denote the exclusion criteria for the further analysis of the framework

SPARQL 1.1) which are the main reason why this framework is also taken into consideration for the implementation and evaluation.

- Apache Jena (Apache, 2016c)

  This prominent Open Source implementation considered the support for relational databases as RDF triple stores deprecated because of the significant performance gains of native RDF triple (e.g. In-Memory or file-based) stores (Apache, 2015). However, due to its wide distribution, its support for SPARQL 1.1 and its Open Source character, it is used as a foundation for various Semantic Web frameworks (e.g. Ontop, FedX, etc.). These are the main reasons why this framework (in particular the API to access SPARQL endpoints) is used for the implementation and evaluation tasks in the following sections.

- Sesame (Sesame, 2016)

  This Open Source Java framework is specially designed for the management of RDF-based data. Especially the parsing and querying capabilities of this framework are promising. However, the support for exposing re-

lational database data as SPARQL endpoints is deprecated in the latest versions (Sesame, 2015). Therefore, only the Sesame API to access SPARQL endpoints is used in the following implementation and evaluation section.

- D2RQ (Cyganiak, 2016)

  This Open Source framework enables a read-only access to relational data exposed as virtual RDF graphs. Thus, no loading or transformation phase to transform relational data to RDF is required. As relational data can be exposed as SPARQL endpoints, these endpoints can be accessed with every SPARQL query engine. A disadvantage is that it only provides preliminary support for SPARQL 1.1, but the possibility to expose relational data as SPARQL endpoints perfectly fits into the FVPD scenario and therefore this framework is used as a foundation for all implementation (cf. Figure 8.5 and Figure 8.6) and evaluation tasks in the following sections.

- Virtuoso (Virtuoso, 2016)

  Virtuoso, which is only partially Open Source, is able to integrate various different data sources like relational databases, file-based storages, etc. However, the integration capabilities are restricted to the commercial version of the framework (Virtuoso, 2016). Above that, Virtuoso as a *Universal Server* (Virtuoso, 2016) only provides access to the data via other APIs like e.g. Jena or Sesame. Therefore, this framework is not considered any further.

- Ontop (Ontop, 2016)

  A promising feature of this Open Source framework is that it already supports R2RML as the mapping language from relations to their RDF-based representations. Moreover, it includes various optimization strategies concerning the join of federated RDF data (Rodríguez-Muro & Rezk, 2015) (which amongst other techniques e.g. concentrate on the concrete join ordering or the removal of unnecessary joins based on table meta data) and it offers preliminary support for SPARQL 1.1 queries. However, in order to exploit these features, a mapping (either OBDA or R2RML) of the RDF-based data to the relational data is required. Moreover, for accessing a SPARQL endpoint, Ontop also heavily relies on the Sesame framework and therefore the implementation and evaluation of Sesame also holds for Ontop.

- Blazegraph (Systap, 2016)

  Blazegraph is also an Open Source implementation of a Semantic Web framework. It also relies on Sesame and offers support for SPARQL 1.1 queries. Similar to FedX is abstracts from Sesame with its own API and this is a promising architecture for further improvements. Hence, this is also considered in the implementation and evaluation in the following sections.

- GraphDB (Ontotext, 2016)

  GraphDB (formerly called OWLIM) is also available as an Open Source implementation. However, the publically available version is restricted to only two simultaneous SPARQL queries. The API of GraphDB shows that this framework can be used with Sesame as well as with Jena for publishing and accessing RDF data. Hence, the implementation and evaluation of Jena and Sesame can easily be transferred to GraphDB and the respective measurements also hold for GraphDB. Hence, this framework is neither implemented nor evaluated separately.

- DARQ (Quilitz, 2006)

  DARQ, as another Open Source implementation, also unites various resources (e.g. databases, files, In-Memory stores, etc.) and gives the impression of working with a single centralized repository. However, the last update was in 2006 and there are no plans to continue or even maintain the implementation (Quilitz, 2006). Hence, this framework is also not considered any further.

More information about the above-mentioned frameworks can be found at their respective websites and a more exhaustive analysis and comparison can be found in (Rakhmawati et al., 2013). Finally, this brief overview lead to the analysis of D2RQ as the basic OBDA mapping and Jena, Sesame, FedX and Blazegraph with respect to their response time in FVPD and in non-partitioned and non-distributed scenarios. Figure 8.5 and Figure 8.6 illustrate the entire approach based on the TPC-W *CUSTOMER* relation for the FVPD as well as for the non-partitioned and non-distributed setup.

## 8.6 Implementation

The following example according to (Das et al., 2012) gives an overview about the concrete transfer of the relational approach to the RDF-based one. The relational

table[17] *CUSTOMER* is stated as follows and for the sake of better readability, illustrated with three rows (i.e. three customer instances).

Table 8.3: *CUSTOMER* Table with 5 Columns

| C_ID | C_FNAME | C_LNAME | C_BALANCE | C_DISCOUNT |
|---|---|---|---|---|
| 1 | Homer | Simpson | 10 | 10 |
| 2 | Marge | Simpson | 20 | 20 |
| 3 | Bart | Simpson | 30 | 30 |

Based on the R2RML mapping, this results in the following RDF triples:

Listing 8.4: R2RML *CUSTOMER* RDF Mapping Example

```
1  PREFIX vocab: http://kohlerjens.de/d2rq customer1/resource/vocab/
2
3  <http://kohlerjens.de/d2rq_customer1/1> rdf:type vocab:CUSTOMER .
4  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_FNAME "Homer" .
5  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_LNAME "Simpson" .
6  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_BALANCE "10" .
7  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_DISCOUNT "10" .
8
9  <http://kohlerjens.de/d2rq_customer1/2> rdf:type vocab:CUSTOMER .
10 <http://kohlerjens.de/d2rq_customer1/2> vocab:C_FNAME "Marge" .
11 <http://kohlerjens.de/d2rq_customer1/2> vocab:C_LNAME "Simpson" .
12 <http://kohlerjens.de/d2rq_customer1/2> vocab:C_BALANCE "20" .
13 <http://kohlerjens.de/d2rq_customer1/2> vocab:C_DISCOUNT "20" .
14
15 <http://kohlerjens.de/d2rq_customer1/3> rdf:type vocab:CUSTOMER .
16 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_FNAME "Bart" .
17 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_LNAME "Simpson" .
18 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_BALANCE "30" .
19 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_DISCOUNT "30" .
```

A SPARQL query that retrieves all *CUSTOMERS* can be formulated as:

Listing 8.5: Non-FVPD SPARQL *CUSTOMER* Query

```
1  PREFIX vocab:http://kohlerjens.de/d2rq_customer/resource/vocab/
2
3  SELECT ?customerID ?fname ?lname ?balance ?discount
4
5  WHERE {
6          ?x vocab:CUSTOMER_C_ID ?customerID .
7          ?x vocab:CUSTOMER_C_FNAME ?fname .
```

[17] the table is shortened for the sake of better readability

```
8           ?x vocab:CUSTOMER_C_LNAME ?lname .
9           ?x vocab:CUSTOMER_C_BALANCE ?balance .
10          ?x vocab:CUSTOMER_C_DISCOUNT ?discount .
11  }
```

The partitioning approach analogous to the one in Chapter 6, can be represented as follows (Table 8.4 and Table 8.5):

Table 8.4: Partition 1 of *CUSTOMER* Table with 5 Columns

| **C_ID** | **C_FNAME** | **C_LNAME** |
|---|---|---|
| 1 | Homer | Simpson |
| 2 | Marge | Simpson |
| 3 | Bart | Simpson |

Table 8.5: Partition 2 of *CUSTOMER* Table with 5 Columns

| **C_ID** | **C_BALANCE** | **C_DISCOUNT** |
|---|---|---|
| 1 | 10 | 10 |
| 2 | 20 | 20 |
| 3 | 30 | 30 |

Accordingly, Listing 8.6 and Listing 8.7 depict the RDB to RDF representation of the FVPD partitions.

Listing 8.6: R2RML *CUSTOMER* Partition 1 RDF Mapping Example

```
1  PREFIX vocab: http://kohlerjens.de/d2rq customer1/resource/vocab/
2
3  <http://kohlerjens.de/d2rq_customer1/1> rdf:type vocab:CUSTOMER .
4  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_FNAME "Homer" .
5  <http://kohlerjens.de/d2rq_customer1/1> vocab:C_LNAME "Simpson" .
6
7  <http://kohlerjens.de/d2rq_customer1/2> rdf:type vocab:CUSTOMER .
8  <http://kohlerjens.de/d2rq_customer1/2> vocab:C_FNAME "Marge" .
9  <http://kohlerjens.de/d2rq_customer1/2> vocab:C_LNAME "Simpson" .
10
11 <http://kohlerjens.de/d2rq_customer1/3> rdf:type vocab:CUSTOMER .
12 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_FNAME "Bart" .
13 <http://kohlerjens.de/d2rq_customer1/3> vocab:C_LNAME "Simpson" .
```

Listing 8.7: R2RML *CUSTOMER* Partition 2 RDF Mapping Example

```
1  PREFIX vocab: http://kohlerjens.de/d2rq customer2/resource/vocab/
2
```

```
<http://kohlerjens.de/d2rq_customer2/1> rdf:type vocab:CUSTOMER .
<http://kohlerjens.de/d2rq_customer2/1> vocab:C_BALANCE "10" .
<http://kohlerjens.de/d2rq_customer2/1> vocab:C_DISCOUNT "10" .

<http://kohlerjens.de/d2rq_customer2/2> rdf:type vocab:CUSTOMER .
<http://kohlerjens.de/d2rq_customer2/2> vocab:C_BALANCE "20" .
<http://kohlerjens.de/d2rq_customer2/2> vocab:C_DISCOUNT "20" .

<http://kohlerjens.de/d2rq_customer2/3> rdf:type vocab:CUSTOMER .
<http://kohlerjens.de/d2rq_customer2/3> vocab:C_BALANCE "30" .
<http://kohlerjens.de/d2rq_customer2/3> vocab:C_DISCOUNT "30" .
```

Based on the original query (Listing 8.5), the FVPD framework creates the separation into 2 *reconstruction queries*. This can be stated as follows:

Listing 8.8: Reconstruction SPARQL 1.0 TPC-W *CUSTOMER* Queries

```
# Query 1:
PREFIX vocab: http://kohlerjens.de/d2rq_customer1/resource/vocab/
SELECT ?customerID ?fname ?lname

WHERE {
        ?x vocab:CUSTOMER_C_ID ?customerID .
        ?x vocab:CUSTOMER_C_FNAME ?fname .
        ?x vocab:CUSTOMER_C_LNAME ?lname .
}

# Query 2:
PREFIX vocab:http://kohlerjens.de/d2rq_customer2/resource/vocab/

SELECT ?customerID ?balance ?discount

WHERE {
        ?x vocab:CUSTOMER_C_ID ?customerID .
        ?x vocab:CUSTOMER_C_BALANCE ?balance .
        ?x vocab:CUSTOMER_C_DISCOUNT ?discount .
}
```

The actual construction of the reconstruction queries is analogous to the approach outlined in the relational approach and illustrated in Figure 8.8.

An XML-file based mapping defines which properties (i.e. attributes in the relational approach) are stored in which FVPD data store. From this XML mapping file, the location of the data stores and the distribution of the properties is derived to create the *reconstruction queries* which are then issued against the

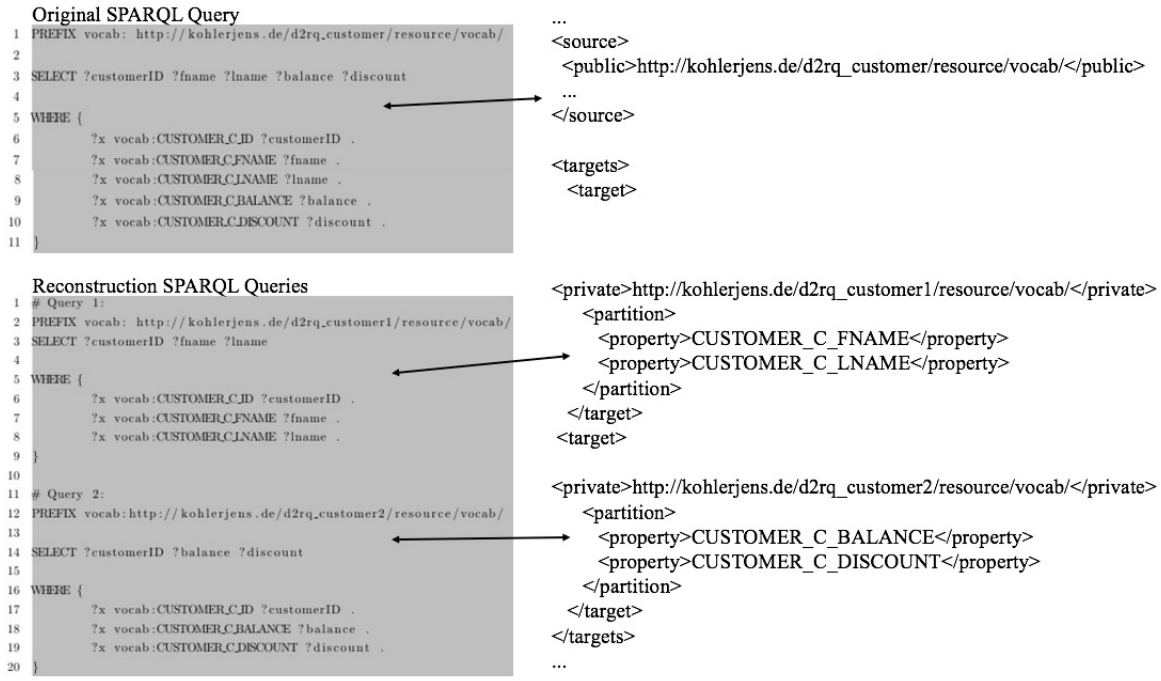



Figure 8.8: Mapping A SPARQL Query to its Corresponding *Reconstruction Queries*

private SPARQL endpoints (cf. Figure 8.6). Thus, the private endpoints remain hidden as they are encoded in a configuration file in the *SeDiCo* framework.

With respect to the comparability of these SPARQL 1.0 queries against a SPARQL 1.1 query, the semantically equal federated SPARQL 1.1 query (with the newly introduced SERVICE keyword) can be stated as follows:

Listing 8.9: Reconstruction SPARQL 1.1 TPC-W *CUSTOMER* Query

```
# Query 1:
PREFIX vocab1:http://kohlerjens.de/d2rq_customer1/resource/vocab/
PREFIX vocab2:http://kohlerjens.de/d2rq_customer2/resource/vocab/

SELECT ?customerID ?fname ?lname ?balance ?discount

WHERE {
        SERVICE <http://kohlerjens.de/d2rq_customer1/sparql> {
                ?x vocab1:CUSTOMER_C_ID ?customerID .
                ?x vocab1:CUSTOMER_C_FNAME ?fname .
                ?x vocab1:CUSTOMER_C_LNAME ?lname .
        }

        SERVICE <http://kohlerjens.de/d2rq_customer2/sparql> {
                 ?x vocab2:CUSTOMER_C_ID ?customerID .
```

```
16                  ?x vocab2:CUSTOMER_C_BALANCE ?balance .
17                  ?x vocab2:CUSTOMER_C_DISCOUNT ?discount .
18          }
19  }
```

Both queries are evaluated in the following section but it has to be mentioned that Listing 8.8 contains 2 queries and the join of the resulting RDF triples has to be performed in the client application logic after the results have been collected. In contrast to this, these steps in Listing 8.9 are already included in the query. In Listing 8.8 it would have been possible to query both endpoints in parallel (e.g. with different programming threads) to improve the overall response time. However, for the sake of comparability with the query in Listing 8.9, this opportunity was neglected.

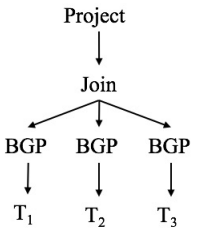
To illustrate the challenging join of federated RDF-based data sets, the transformation from a SPARQL to an SQL query is now briefly illustrated based on a simplified query similar to the previous listings. Such a SPARQL query could be stated as follows:

Listing 8.10: SPARQL to SQL Query Example

```
1  PREFIX vocab: http://kohlerjens.de/d2rq_customer1/resource/vocab/
2
3  SELECT ?customerID ?fname ?lname
4
5  WHERE {
6          ?x vocab:CUSTOMER_C_ID ?customerID .
7          ?x vocab:CUSTOMER_C_FNAME ?fname .
8          ?x vocab:CUSTOMER_C_LNAME ?lname .
9  }
```

This query results in a query tree as depicted in Figure 8.9[18].

[18]adapted from (Rodríguez-Muro & Rezk, 2015)




with

$T_1$ = (?x vocab:CUSTOMER_C_ID ?customerID .)
$T_2$ = (?x vocab:CUSTOMER_C_FNAME ?fname . )
$T_3$ = (?x vocab:CUSTOMER_C_LNAME ?lname . )

Figure 8.9: SPARQL to SQL Example - Query Tree

Here in this nave representation, the challenging character of the join can be derived, as for every matching triple (basic graph pattern, BGP) $T_i$ a join has to be performed. Hence, this results in the following SQL query[19]

Listing 8.11: SPARQL to SQL Query Example - The SQL Query

```
1  SELECT DISTINCT
2          T1_CUSTOMER.C_ID,
3          T2_CUSTOMER.C_FNAME,
4          T3_CUSTOMER.C_LNAME,
5
6  FROM
7          CUSTOMER AS T1_CUSTOMER,
8          CUSTOMER AS T2_CUSTOMER,
9          CUSTOMER AS T3_CUSTOMER,
10
11 WHERE (
12         T1_CUSTOMER.C_ID = T2_CUSTOMER.C_ID AND
13         T1_CUSTOMER.C_ID = T3_CUSTOMER.C_ID AND
14         T2_CUSTOMER.C_FNAME IS NOT NULL AND
15         T3_CUSTOMER.C_LNAME IS NOT NULL
16 )
```

This results in a join for every BGP or in other words for every triple that is defined in the SPARQL query. Therefore, current optimization strategies focus on optimizing the order of the joins (e.g. smaller relations first to avoid joins, etc.).

[19]The query tree for the other partition that contains the *balance* and the *discount* is analogously and therefore not illustrated. Eventually, this query tree also would have to be executed and the results of both trees would have to be joined for the final result set.

## 8.7 Evaluation

As now the theoretical background and the implementation for the entire approach is outlined, the evaluation measures the response time in concrete numbers in the following section.

### 8.7.1 Evaluation Environment

For the evaluation of the before-mentioned Semantic Web frameworks and the underlying FVPD approach, the same evaluation environment as outlined in Section 6.1 was used. Accordingly, the same data set, namely the TPC-W *CUSTOMER* relation with a maximum of 288K rows was applied. Exposing this data set as SPARQL endpoints result in the transformation of this data in virtual RDF graphs through OBDA mappings. As the *CUSTOMER* relation contains 19 attributes the virtual RDF graph contains 5,472 million (288K rows * 19 attributes) as a maximum number of triples. Compared to average LOD repositories like DBPedia which contains ∼7 billion triples this is a relatively small number. However, it illustrates the scalability of FVPD RDF-based data and facilitates the direct comparison against traditional relational database implementations.

The central metric for this evaluation is also similar to the evaluation in Section 6.1 and similarly, all queries were executed 3 times and the average response time is denoted in the tables of this section. The response time in this section also includes an iteration through the result sets, as they are transferred to the client as data streams (unlike in e.g. JDBC which uses Java objects) and this stream has to be written (*deserialized*) in the corresponding Java objects in order to remain comparable to the ORM-based evaluation in Chapter 6. Additionally, this approach is also used in other benchmarks such as e.g. in (Haase et al., 2010).

Furthermore, all endpoints were accessed via SPARQL queries embedded in their corresponding Java APIs. Another issue that deals with the evaluation of accessing data in federated RDF repositories is the distinction between a local and a remote federation. This is also similar to the evaluation setup in Chapter 6. Accordingly, in LOD scenarios this is a challenging task when RDF-based data are stored in dynamically scalable Cloud Computing architectures. Not only the network overhead (available bandwidth and network latency over the Internet, etc.) and unknown endpoint implementations with different SPARQL or RDF versions but also frequently changing data (e.g. DBPedia which uses Wikipedia

as foundation) lead to unpredictable behaviors (Montoya et al., 2012). Hence, in real-world scenarios with different SPARQL endpoints involved, it might not be clear which endpoint supports which SPARQL or RDF version, neither is it transparent which endpoint defines which SLAs (e.g. maximum amount of triples returned per query, etc.) and how they are negotiated between the endpoint provider and the querying client (Buil-Aranda et al., 2014).

Transferred to the federated character of the FVPD approach, these factors also become crucial when different cloud vendors, implementations and SLAs are involved. Hence, evaluations based on remote environments have to be considered carefully, as their reproducibility might not be given. In order to minimize these factors, the same experimental setup as in Section 6.1 with a *local* and a *remote* implementation (with private clouds to which only the author has access to) is used in this evaluation. This differentiation can also be found in other benchmark papers such as e.g. in (Schwarte et al., 2012). Now, the following tables illustrate the respective response times starting with measuring the *local* environment which is then followed by the *remote* one.

### 8.7.2 Local SPARQL 1.0 Evaluation

This section illustrates the measured values for the *local* non-FVPD (Figure 8.10) and the FVPD-based (Figure 8.11) evaluation.

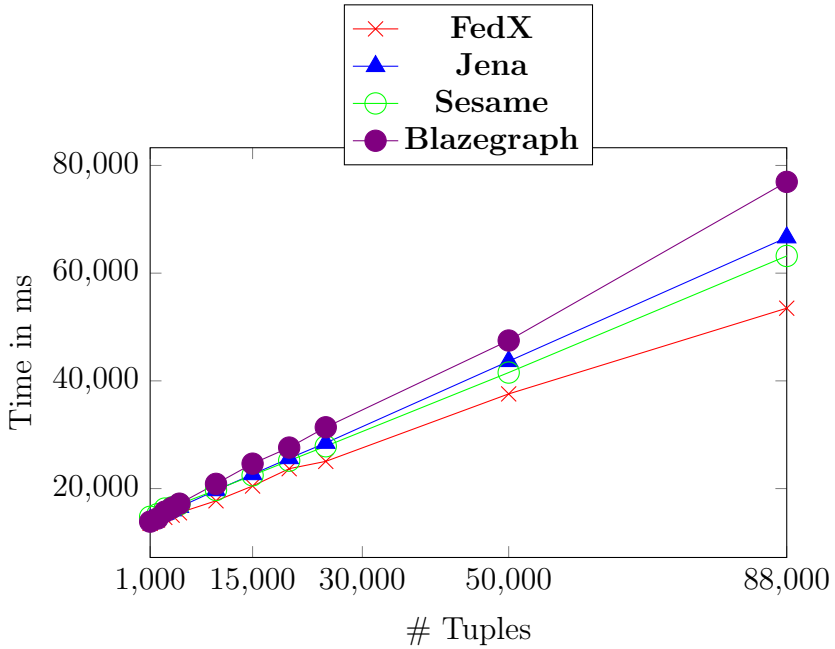



Figure 8.10: Local Non-FVPD OBDA Framework Evaluation

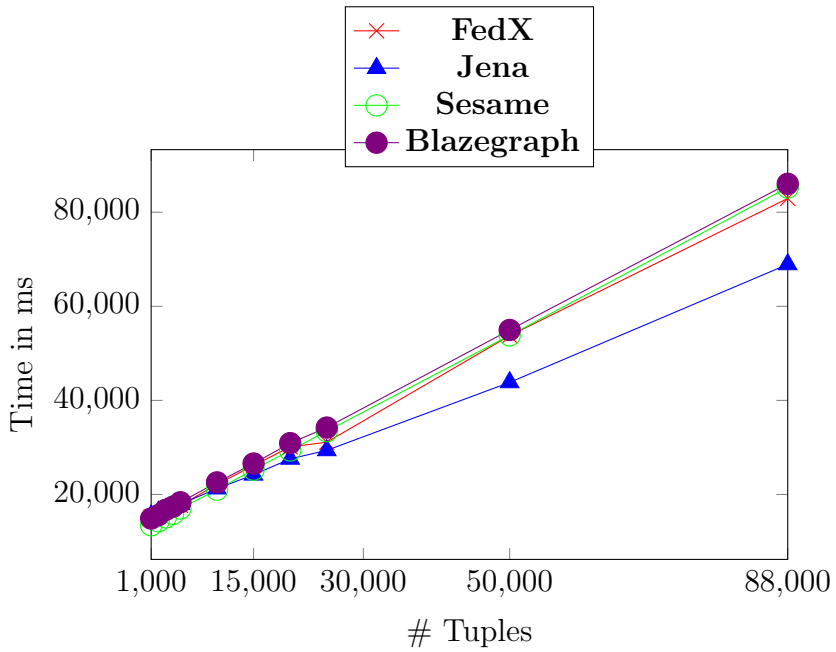



Figure 8.11: Local FVPD OBDA Framework Evaluation

### 8.7.3 Remote SPARQL 1.0 Evaluation

Accordingly, this section illustrates the measured values for the *remote* non-FVPD (Figure 8.12) and the FVPD-based (Figure 8.13) evaluation.

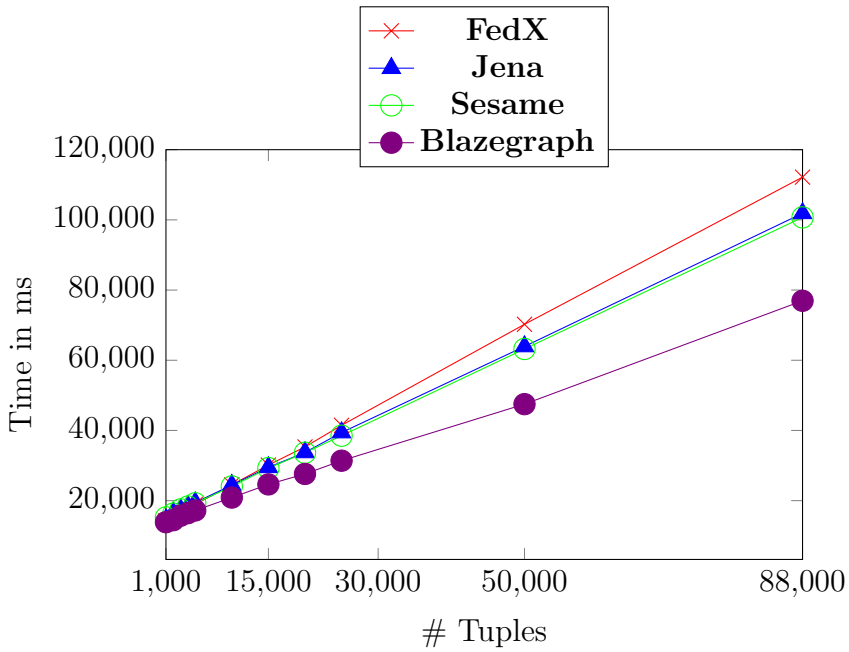



Figure 8.12: Remote Non-FVPD OBDA Framework Evaluation

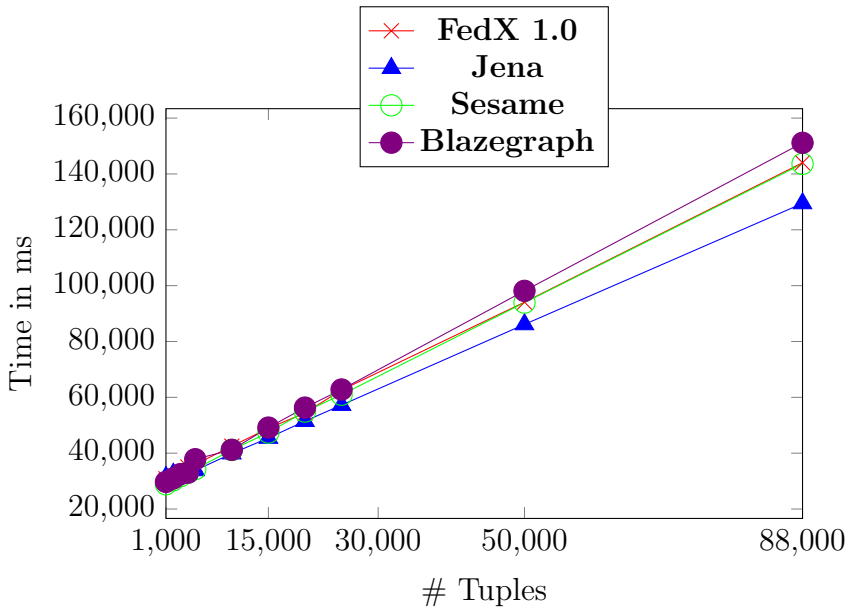



Figure 8.13: Remote FVPD OBDA Framework Evaluation

### 8.7.4 Local and Remote SPARQL 1.1 Evaluation

This section illustrates the local and the remote measurement of the SPARQL 1.1 query, illustrated in Listing 8.9. Here, the measurements of the previous sections are repeated, except that a SPARQL 1.1 query was used. It has to be mentioned that currently, FedX is the only framework that has the SPARQL 1.1 standard implemented yet and therefore the other frameworks could not be investigated here.

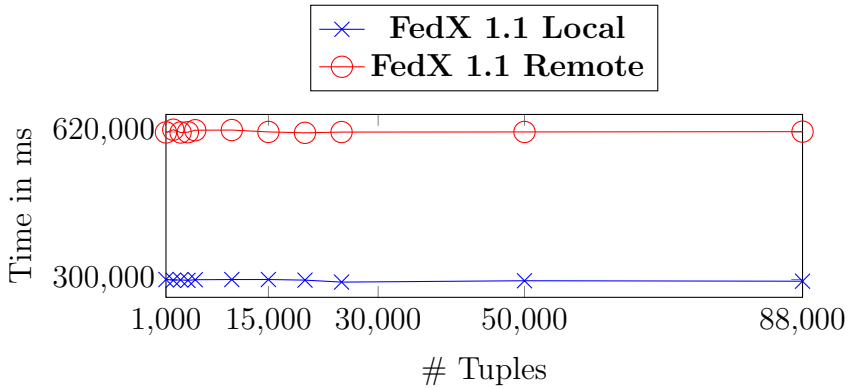



Figure 8.14: Local and Remote FVPD OBDA SPARQL 1.1 Framework Evaluation

## 8.8 Conclusion

To conclude the evaluation, it can be noted that the results measured in (Haase et al., 2010) could not be confirmed. Surprisingly, the response time for the *local* setup over the evaluated federation is similar (except for FedX) compared to the single endpoint implementation with a non-partitioned and non-distributed data set (Figure 8.10 compared to Figure 8.14). Although, in the *remote* setup the average response time degrades by factor ∼2 (Figure 8.12 compared against Figure 8.13), it can be noted that the federation does not have such an high impact on the average performance as it was elaborated in (Haase et al., 2010). However, it has to be mentioned that in this evaluation a comparatively small data set was used compared to e.g. DBPedia (∼7 billion triples) or ∼110,000K triples in (Haase et al., 2010). Moreover, the evaluation in this section focused on an federation of only 2 different SPARQL endpoints, whereas the evaluation in (Haase et al., 2010) focused on 12 different endpoints. Hence it must be assumed that with a growing number of triples and a growing number of partitions (or federations) the performance degrades will also increase. Here, further measurements that take a problem-driven data distribution (with respect to the number of vertical partitions), a corresponding number of different endpoints and a use-case driven data volume into consideration are necessary.

(Haase et al., 2010) determined that complex queries are more effected by performance losses (averagely by factor ∼3) than simple queries. This could be confirmed in this evaluation with the separated view of SPARQL 1.0 (simple queries) and SPARQL 1.1 (complex queries). Here, the figures above show that the simple SPARQL 1.0 queries outperform the complex SPARQL 1.1 queries averagely by factor ∼10 for both, the *local* (Figure 8.11) and the *remote* (Figure 8.13) setup. Finally, the response time for the SPARQL 1.1 query does not depend on the number of queried triples: it is almost constant in Figure 8.14. It is dependent on the size of the entire data store, i.e. the search space. This means that the query first reconstructs the entire data store and then after this reconstruction, the query parameters are evaluated. Hence, the reconstruction phase can be identified as the crucial factor for the performance degrade and so, especially complex queries could benefit from further optimizations with respect to this expensive operation.

Comparing the measured *RDF-based* results against the *SSD-based* TPC-W benchmark results in Section 6.6 shows that the average response time degrades

by factor ∼22 in the *local* non-FVPD setup. Interestingly, in the *local* FVPD setup, the performance degrades only by factor ∼5. Moreover, the measurements for the *remote* setup are almost similar with performance degrades by factor ∼28 for the non-FVPD and by factor ∼4 for the FVPD data set. Thus, on the one hand, compared to a traditional database access via JDBC or via ORM, there is a significant performance loss that has to be considered in real-world scenarios. On the other hand however, accessing RDF-based data offers a schema free and more flexible way of querying data or even finding unknown relations, links or other information between data with the usage of inference engines.

Contrasting the *local* and the *remote* figures above, it can be concluded that although the integration of the FVPD approach decreases the response times by averagely factor ∼2 (in the *remote* setup)[20], it is a viable approach if further optimizations are taken into consideration. Such optimizations could involve strategies and techniques depicted in Section 4.2 and Section 4.1 of this work, namely *caching* or *query rewriting*. Transferring such concepts to the proposed FVPD approach results in both, faster response times and a privacy and security-aware way of storing such data in RDF-based form.

To sum up the conclusion with respect to the hypothesis expressed in the problem formulation of Section 8.2, it can be stated that it can be verified. Finally, the response times are in the same order of magnitude as the results in this section show. Hence, for the given scenario the FVPD approach with data exposed as SPARQL endpoints is a feasible approach with respect to the response times. Above that, the FVPD approach applied to RDF-based data offers the same data security and privacy-enhancing capabilities (*Security-by-Distribution*) as already shown in Section 2.2.

## 8.9 Outlook and Future Work

The evaluation results depicted in the previous section show that there is an enormous performance improvement potential for querying FVPD RDF-based data based on relational databases as storage engines. Here, advanced approaches like caching triples (e.g. with NoSQL In-Memory architectures) as proposed in (Martin et al., 2010) or in (Cudre-Mauroux et al., 2013) or as applied in Section 5.2 of this work are promising. However, caching triples requires new cache coherence

[20] whereas in the *local* setup the performance is similar

protocols, especially for the invalidation of triples. Such approaches could benefit from the long history of caching. However, only real implementations and their evaluation in real-world scenarios will prove the feasibility of such triple-based cache coherence protocols. With further respect to this (Cudre-Mauroux et al., 2013) conducted an evaluation of the 4 basic NoSQL architectures (key-value, column, document and graph stores), but an architecture that clearly outperformed all others could not be determined. Interestingly, they elaborated that all NoSQL architectures are able to compete against native RDF stores with respect to their response time. However, it has also to be mentioned that manipulating RDF data (and the corresponding caches) was not part of their experiments, and moreover, the application of the FVPD approach on native triple stores is also an interesting future work task.

Closely aligned to caching, in particular to the cache warming phase is prefetching data. Here (Lorey & Naumann, 2013) demonstrated an interesting approach based on previously issued SPARQL queries. Transferring this approach to the RDF-based FVPD setup might be a promising future work task.

Another approach which is taken into consideration in the future development of this RDF-based FVPD approach is *Result Ranking* (Görlitz & Staab, 2011). Although SPARQL (like SQL) returns results in unordered sets (unless *ORDER BY* is used), triple ranking algorithms like (Franz et al., 2009) (Ning et al., 2008) might be helpful in use cases where not all but only parts of the results are required. Reducing the result set to only highly ranked triples would also reduce the number of required join operations and thus, it would be possible to transfer smaller result sets to the querying client faster.

Further future work tasks could also involve a detailed analysis and evaluation of join algorithms. Recent approaches, besides the nested-loops, hash and sorted-merge join outlined in Section 5.2 use the *MapReduce* framework for the join of federated SPARQL queries (Gimenez-Garcia et al., 2014). This is considered an appealing approach, especially when RDF data is serialized as XML, JSON or other files. For MapReduce this is advantageous because its most prominent Hadoop implementation uses a distributed file system, the Hadoop File System (HDFS) that is able to handle serialized RDF triples in parallel and thus very efficiently. With respect to this, (Gimenez-Garcia et al., 2014) provide useful information on how such a MapReduce-based approach can be implemented. Therefore, transferring these approaches the FVPD approach in order to improve

the response time are considered as promising future work tasks for the further ongoing *SeDiCo* framework development.

# Summary and Outlook

This chapter now concludes and summarizes the entire thesis, its approach and its results. It also provides an outlook about further research works with respect to the *SeDiCo* framework implementation.

## Summary

In recent years the interest in Cloud Computing has grown significantly as e.g. Gartner (Gartner, 2013) or IDC (Gens & Shirer, 2013) regularly show. There are several reasons for this: the avoidance of large initial investments in hardware infrastructures, the shift of maintenance, update or upgrade tasks towards cloud providers, the universal accessibility via Internet to cloud offers, data synchronization between various heterogeneous (even mobile) devices, etc. As databases (and especially relational databases still are) are the foundation for a variety of applications, storing these data volumes in a cloud infrastructure would mean a great benefit for enterprises and for end customers. However, data security and privacy issues prevent enterprises as well as end customers from using especially public cloud infrastructures for the management of their sensitive data.

The *SeDiCo* framework, developed by the author of this work and the student works listed below, addresses these concerns with a vertical database partitioning and distribution (so-called FVPD) approach, that splits database relations and distributes the chunks across (ideally) different clouds. Moreover, the framework aims at avoiding vendor lock-ins with respect to the used database systems and with respect to the cloud providers through the usage of database abstraction with Hibernate as an Object-relational Mapper (ORM) and through cloud abstraction with jclouds as a cloud abstraction layer on the IaaS service layer.

Although *SeDiCo* increases the level of security and privacy, it leads to tremendous performance losses, as the FVPD data have to be joined together again before they are actually accessed. Despite the technological feasibility, the framework was not usable in practical usage scenarios due to the tremendous performance degrade. Hence, this thesis focused on this performance challenge and conceptualized, implemented and evaluated *query mechanisms* for the FVPD partitions. For this, the research problem was defined as a minimization problem with respect to the response time. Moreover, the entire FVPD approach was formalized according to Codds relational model (Codd, 1970).

In a second step, the current state-of-the-art regarding the key concepts (i.e. *Security* and *Privacy*, *Cloud Computing*, *Database Abstraction (ORM)*, *Query Rewriting*, *Caching* and *Benchmarking*) were elaborated. Afterwards, the current state-of-the-art of the *SeDiCo* framework was developed and illustrated. Based on this, 3 query mechanisms were conceptualized and developed. Accordingly, the query mechanisms were evaluated against a non-partitioned and non-distributed data set as well as against *SeDiCo*s FVPD implementation. The evaluation, based on the TPC-W benchmark showed that *query rewriting* and *caching* improve the caching compared to the initial *SeDiCo* implementation by factors. Surprisingly, both query mechanisms improved the response time to a level that is in the same order of magnitude as queries against a non-distributed and non-partitioned setup.

Moreover, this thesis showed that a combination of the query mechanisms would also a viable approach to achieve additional performance gains. The evaluation of this sustainable idea is in the focus of the future work regarding the *SeDiCo* development. This leads to the final conclusion that the developed query mechanisms have a remarkable impact on the response time and that this work serves as a guideline for interested practitioners and shows which query mechanism promises which performance gains. Another aspect is the fact that the field of database research is not bound to a concrete application domain and as database workloads cannot be generalized, the *SeDiCo* framework has to be tested and evaluated against various domains and application scenarios. Therefore, this thesis developed a basic performance metric, which can be used to test and evaluate further scenarios and applications and to determine how the FVPD approach affects the response time. The thesis also contains an analysis of the usage of the framework in a Semantic Web application scenario in which the FVPD approach is applied. The preliminary response time evaluation results are promising, however, further optimization strategies have to be elaborated, applied and evaluated in

order to reach similar performance results with traditional relational database queries.

## List of Publications Related to the Thesis

This section summarizes the author's publications and their thematic focus to the respective thesis chapters. It can be noted that central aspects and ideas of all chapters have been successfully published and presented at national and international conferences and journals. Thus, this thesis added substantial extensions to them and integrated and structured the publications in a central and thematically focused framework.

| **No.** | **Publication** | **Summary** | **Ref. to Thesis Chapter** |
|---|---|---|---|
| 1 | Kohler J.; Specht T.; Simov K.: An Approach for a Security and Privacy-Aware Cloud-Based Storage of Data in the Semantic Web. In: Proc. of The First IEEE International Conference on Computer Communication and the Internet (ICCCI 2016). Wuhan, China. | This paper conceptualizes, implements and evaluates the FVPD approach for RDF-based data and evaluates a TPC-W benchmark scenario with SPARQL queries based on *customer* data. Hence, this thesis added a more detailed analysis of different Semantic Web Frameworks and different benchmark frameworks. Moreover, this thesis added a complexity analysis and a correctness proof of the proposed approach. Moreover, it introduces the selection criteria for the Semantic Web Frameworks and the benchmarks in closer detail and describes how the respective frameworks and benchmarks were selected. All in all, this paper provides a first performance evaluation in the Semantic Web and gives an outlook about further works with respect to native triple store implementations. | 8 |

| | | | |
|---|---|---|---|
| 2 | Werner S., Kohler J.; Specht T.; Simov K.: Cache Synchronization in a Vertically Distributed Cloud Database Environment. In: Proc. of AKWI 2016 - Arbeitskreis Wirtschaftsinformatik an Fachhochschulen, September 2016. Brandenburg, Germany. | This paper analyses the challenging synchronization task when different client-based caches read and write data to and from relational FVPD data. Consistency is a major topic here, this paper evaluates ACID (Atomicity, Consistency, Isolation, Durability) as hard consistency (ACID) and BASE (Basically Available Soft State Eventual Consistency) as a weaker form of consistent data. The evaluation shows that is a distributed client-based caching approach hard ACID consistency is not applicable from a performance point of view. Here, weaker forms (in between ACID and BASE), dependent on the required consistency are more promising models. | 4, 5, 6 |
| 3 | Kohler, J.; Simov, K.; Specht, T.: Analysis of the Join Performance in Vertically Distributed Cloud Databases. In: International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS), 6(2), 2016 | This journal paper outlines the whole *SeDiCo* approach in more detail from an architectural point of view. It discusses different join locations (application, database driver, ORM layer, etc.) where the time-consuming join in the FVPD approach could be performed efficiently. This analysis further justifies the usage of an ORM for the FVPD approach and presents a detailed performance analysis of the pure *SeDiCo* framework usage, based on the TPC-W benchmark. These benchmark results are tested against a real world application scenario that uses the FVPD approach in a Wikipedia user database to guarantee a higher level of anonymization and to implement the databases securely in distributed public cloud architecture. | 1, 2, 3, 4, 5, 6 |
| 4 | Kohler J.; Specht T.: Analysis of Cache Implementations in a Vertically Distributed Cloud Data Store. In: Proc. of The 3rd IEEE World Conference on Complex Systems, November 2015. Marrakech, Morocco. | The aim of this work was a deeper analysis of the caching approach with an evaluation of different cache storages. Here, an In-Memory key-value store, a locally installed relational database and a JSON file-based cache implementation were used and the response time of all storages was measured and compared. This work proved that for the FVPD approach the In-Memory cache outperforms the other ones, however they also have their raison dtre when other requirements such as non-volatile storage, larger cache memories or strong ACID support, etc. are more important than pure response time. | 4, 5, 6 |

| | | | |
|---|---|---|---|
| 5 | Kohler J.; Simov K.; Fiech A.; Specht T.: On The Performance Of Query Rewriting In Vertically Distributed Cloud Databases. In: Proc. of The International Conference Advanced Computing for Innovation ACOMIN 2015, November 2015. Sofia, Bulgaria. | The focus of this work was the conceptualization, formalization and a first prototypical implementation followed by an evaluation of the query rewriting approach. The presented thesis extends this work now with a more extensive evaluation that also takes different cloud infrastructures into concern. This work proved the technological feasibility of the query rewriting approach and the evaluation in this work proved that the approach is worth to be pursued further as a foundation for this thesis. | 1, 2, 3, 4, 5, 6 |
| 6 | Kohler J.; Specht T.: Dynamic Software-Based Scaling In Private Clouds. In: Proc. of AKWI 2015 - Arbeitskreis Wirtschaftsinformatik an Fachhochschulen, September 2015. Luzern, Switzerland. | This is a supplementary work in which a dynamic, automated, software-driven, horizontal scalability mechanism for IaaS clouds was developed. On this layer, dynamic scalability is still challenging, as infrastructures (e.g. entire virtual machines) with a heterogeneous software stack have to be scaled. Due to this enormous heterogeneity these infrastructures have to be scaled manually, which is a daunting task. Due to the missing data encryption and other security challenges, *SeDiCo* mainly focuses on databases on the IaaS layer and thus this work contributed to the overall *SeDiCo* framework development with a dynamic and automated software scalability layer. | 3 |

| | | | |
|---|---|---|---|
| 7 | Kohler J.; Specht T.: A Performance Comparison Between Parallel And Lazy Fetching in Vertically Distributed Cloud Databases. In: Proc. of The International Conference on Cloud Computing Technologies and Applications - CloudTech 2015, June 2015. Marrakesh, Morocco. | This work investigates 2 different data fetching strategies that are similarly offered by ORMs: lazy and eager fetching. Eager fetching in the context of this work is called parallel fetching as with respect to the FVPD approach, eager and lazy fetching result in the same implementation. Here, parallel fetching collects query-matching rows from all partitions in parallel threads to improve the response time, whereas lazy fetching collects rows from one partition and queries only the corresponding partitions if the row is really accessed by the client eventually. This work was conducted during the investigations of the caching approach and this thesis now concludes that both strategies are in the same order of magnitude as non-partitioned and non-distributed queries are, however slightly slower. This work shows that the parallel outperforms the lazy strategy at cost of higher requirements for the querying clients (i.e. processors that support multiple threads). | 4, 5, 6 |
| 8 | Kohler J.; Specht T.: Performance Analysis of Vertically Partitioned Data in Clouds Through a Client-Based In-Memory Key-Value Store Cache. In: Proc. of The 8th International Conference on Computational Intelligence in Security for Information Systems, June 2015. Burgos, Spain. | This work continues the research work concerning the introduction of a client-based cache between the clients and the databases in the clouds. Here, the integration of an In-Memory cache (i.e. memcached) that holds recently accessed data in order to improve the overall insert, update and delete performance is presented. This work also shows that an In-Memory cache is an appealing way to achieve substantial performance improvements. The performance evaluation of the cache implementation in this paper shows that the data access times are comparable with the access times of non-partitioned data. However, the introduction of a cache involves adequate loading, synchronization and invalidation strategies, which are only briefly sketched to not exceeding the limits of the paper. | 1, 2, 3, 4, 5, 6 |

| | | | |
|---|---|---|---|
| 9 | Kohler J.; Specht T.: Vertical Query-Join Benchmark in a Cloud Database Environment. In: Proc. of The 2nd World Conference on Complex Systems, November 2014. Agadir, Morocco. | The focus of this paper is the performance comparison of queries between vertically distributed and non-distributed data. Foundation for the basic measurement of the required response time was a locally installed non-distributed database table. This table was then vertically distributed and the query (i.e. select * from table) was performed again. In the basic setup, the database and the manipulating client were installed on the same physical hardware. Afterwards, the query measurement was repeated with a 1 GBit network connection between the manipulating client and the databases in different clouds. Thus, as the result of the paper, the required response times were recorded and compared. A tremendous performance loss was discovered and caching approaches to overcome these performance losses were sketched. | 1, 2, 3, 4, 5, 6 |
| 10 | Kohler J.; Specht T.: Analysis of the Join-Problem in Vertically Distributed Databases (in German). In: Proc. of AKWI 2014 - Arbeitskreis Wirtschaftsinformatik an Fachhochschulen, September 2014. Regensburg, Germany. | This work concentrates on the introduction of a cache between the cloud databases and the clients in order to overcome the performance loss caused by the vertical distribution approach. Basically, there are three approaches for a cache: an In-Memory cache, inspired by the new trend topic In-Memory databases, a cache in form of a locally installed intermediate database and an intermediary file-based JSON cache. This work analyses the pros and cons of all 3 caching architectures. The results of the analysis show, that there is no optimal architecture, as the cache architecture always depends on the underlying database workload. Nevertheless, it creates a valuable foundation for the implementation work and the subsequent performance evaluation of the presented thesis. | 1, 2, 3, 4 |

| | | | |
|---|---|---|---|
| 11 | Velikova D.; Kohler J.; Gerten R.: Case Study On Financing And Business Development Processes In Technopreneurship. In: Proc. of European Conference on Innovation and Entrepreneurship (ECIE) 2014, September 2014. Belfast, Ireland. | This case study investigated the reason for the remarkable innovation gap between European countries. Despite various innovation and research measurements like Horizon 2020, FP7 and FP6, in 2013 countries like Sweden, Denmark and Germany were presented as innovation leaders, whereas countries like Romania or Bulgaria were identified as modest innovating countries. The hypothesis of this paper was the assumption that the different funding and business development processes of the countries are a key issue for the innovation gap. The case study of three different venture capitalists (a Bulgarian, a German and one from Switzerland) not only confirmed the hypothesis but also provided useful information and approaches on how to overcome the gap in the near future. Therefore, on the one hand, organizational changes in the funding and business development processes are necessary. On the other hand, Cloud Computing could serve as a technological architecture, to develop, enable, support and accelerate collaboration and cooperation activities of all EU member countries. | 2 |
| 12 | Kohler J.; Specht T.: Vertical Update-Join Benchmark in a Cloud Database Environment. In: Proc. of WiWiTa 2014. Wismarer Wirtschaftsinformatiktage. June 2014. | The focus of this paper is the update performance comparison between vertically distributed and non-distributed data. Foundation for the measurement of the required time to manipulate data was a locally installed non-distributed database table. This table was then vertically distributed and the same manipulations were performed again. In the basic setup, the database and the manipulating client were installed on the same physical hardware. Afterwards, the measurement was repeated with a 1 Gbit network connection between the manipulating client and the databases in different clouds. Thus, as the result of the paper, the required manipulation times were recorded and compared. Finally, a tremendous performance loss was discovered and approaches to overcome these performance losses were sketched. | 4, 5, 6 |

| | | | |
|---|---|---|---|
| 13 | Kohler J.; Specht T.: A Marketplace for the Cloud: Comparison of Data Stores Through QoS/SLA Mapping. (in German) In: Technologien fuer digitale Innovationen. Springer Verlag 2014. Wiesbaden, Germany. | This paper deals with the challenge of finding an adequate cloud vendor and the lack of comparable cloud offerings. As there is no defined structure, finding suitable cloud offerings is a daunting task. This work proposes a unified structure for cloud offerings (i.e. for their functional and non-functional criteria) and presents a mathematical model to compare them semi-automatically. | 3 |
| 14 | Mueller P.; Kohler J.; Specht T.: A Vertical Data Distribution Approach in the Cloud. (in German) In: eJournal of AKWI - Arbeitskreis Wirtschaftsinformatik. Februar 2014. Luzern. ISSN: 2296-4592. http://akwi.hswlu.ch | This paper focuses on the data integration after their distribution. The entire *SeDiCo* approach is based on the vertical distribution of data, but joining data together afterwards is as important as their distribution. This paper analyzes five possible locations where distributed data could be joined. There are various possibilities directly on the database layer (i.e. database links, trigger, log file analysis, etc.) or one layer above in the database drivers. Furthermore, this work analyzes object-relational mappers (i.e. Hibernate or the Java Persistence API) as a possible location for the join. Lastly, a join in the application layer (i.e. Java Beans) is taken into consideration. Moreover, these locations not only are analyzed but also evaluated. This evaluation showed that a join in the object-relational mapper (Hibernate) is the most appropriate location with respect to the support of different database vendors in the *SeDiCo* framework. | 1, 2, 3, 4, 5, 6 |

| | | | |
|---|---|---|---|
| 15 | Kohler J.; Specht T.: A Marketplace for an Efficient and Transparent XaaS-Evaluation. (in German) In: Proc. of AKWI - Arbeitskreis Wirtschaftsinformatik an Fachhochschulen, September 2013. Friedberg, Germany. | This work points out another great and still open challenge in Cloud Computing that refers to the vendor lock-in. This problem stems from different vendor interfaces (APIs) and from different service offerings. Due to the fact that every provider has its own programming interfaces and its own structure to offer services, it is a complex and expensive task to exchange a cloud vendor with another one. Thus, this is a challenging task throughout all cloud architectures (IaaS, PaaS and SaaS). This paper includes a literature analysis of Quality of Service (QoS) and Service Level Agreement (SLA) definitions from service-oriented architectures (2002) to the current Cloud Computing hype (2013) and extracts 25 key criteria, that every service should include, in order to adequately describe a service offering. | 3 |
| 16 | Kohler J.: *SeDiCo* - Towards a Framework for a Secure and Distributed Cloud Data Store. In: Proc. of Chip-To-Cloud Security Forum, September 2012. Nice, France. | This paper stresses data security and data protection as the main problems of Cloud Computing. It visualizes the fundamental approach of the *SeDiCo* Framework and presents the vertical data partitioning distribution approach as an idea to overcome data security and protection problems. It also conceptualizes the entire framework and provides an overview about upcoming challenges for its implementation, i.e. different vendor interfaces, performance considerations and ACID-conform transactions. | 2 |

Above that, the works described above contributed to the current state-of-the-art with the following citations:

Kohler J.; Specht T.: Vertical Query-Join Benchmark in a Cloud Database Environment. In: Proc. of The 2nd World Conference on Complex Systems, November 2014. Agadir, Morocco.

*Cited by:*

Awadh, A.: Distributed relational database performance in Cloud Computing: an investigative study. Master Thesis at Auckland University of Technology. 2015. Auckland, Australia

Kohler J.; Simov K.; Fiech A.; Specht T.: On The Performance Of Query Rewriting In Vertically Distributed Cloud Databases. In: Proc. of The Interna-

tional Conference Advanced Computing for Innovation ACOMIN 2015, November 2015. Sofia, Bulgaria.

*Cited by:*

Kaur, K.; Laxmi, V.: Partitioning Techniques in Cloud Data Storage: Review Paper. In: Shrimali, T. (Eds.) International Journal of Advanced Research in Computer Science. Vol. 8(5). 2017. India.

The entire *SeDiCo* framework development was funded by the following institutions:

- Federal Ministry of Education and Research, Germany
- MFG Foundation Baden-Wuerttemberg, Karl Steinbuch Research Program, Germany

## List of Theses Supervised by the Author

The following list presents all student works, supervised by the author, that were developed in the context of the *SeDiCo* framework. These works represent small development and testing tasks that were helpful for the author to implement and evaluate the FVPD approach.

1. Seminar Paper 06/2016

   Lorenz, Richard: Conceptualization, Implementation and Evaluation of a Vertical Partitioning Approach for NoSQL Document Stores. (in German)

2. Bachelor-Thesis 09/2015

   Taenzer, Martin: Trusted Cloud: A Way Towards a Secure and Trustworthy Cloud. (in German)

3. Bachelor-Thesis 09/2015

   Werner, Stefan: Conceptualization, Implementation and Evaluation of Cache Synchronization Mechanisms in a Vertically Partitioned Cloud Database Application. (in German)

4. Bachelor-Thesis 08/2015

   Heiler, Daniel: Parallel Data Access Through Query-Rewriting in a Vertically Distributed Cloud Database Environment

5. Bachelor-Thesis 07/2015

   Atilgan, Tunahan: Evaluation of Semantic Service Registries for Web Services. (in German)

6. Bachelor-Thesis 05/2015

   Sahin, Huzeyfe: Integration of a Middle-tier Database Cache into a Vertically Partitioned Cloud Architecture. (in German)

7. Bachelor-Thesis 05/2015

   Schmidt, Sonny: Conceptualization and Implementation of an Automated Horizontal Scaling Platform for Cloud-based Database Cluster. (in German)

8. Bachelor-Thesis 01/2015

   Eslengert, Igor: Conceptualization and Implementation of a Web-based and Automated Cloud Service Level Agreement Directory for the IaaS Layer. (in German)

9. Bachelor-Thesis 10/2014

   Hlipala, Christof: Conceptualization and Implementation of a Dynamic Scaling Mechanism for CloudStack. (in German)

10. Bachelor-Thesis 07/2013

    Mueller, Patrick: Vertical Data Partitioning in a Distributed Cloud Architecture. (in German)

11. Bachelor-Thesis 07/2013

    Kaiser, Leon: Framework Evaluation of Cloud Abstraction Layers for the Integration of Distributed Data. (in German)

## Approbation of the Results

**Research Papers.** The above-mentioned list of the author's publication is the result of research papers that were created and published before and during the course of this thesis. All published papers include the attendance and the presentation of the work at the respective conference by the author. The thesis added substantial extensions to these works as outlined in more detail below. Based on the promising results of this thesis further research papers are planned

with respect to the topics mentioned in the outlook. Concrete planned works focus on the adaption of the FVPD approach to NoSQL databases (key-value, column, document, and graph stores).

**Student Theses.** The same holds for the above-mentioned list of theses supervised by the author. These seminar and bachelor theses were conducted during the course of the thesis to investigate further topics that are related (but not in the central focus) to the thesis. Similarly, further student works (and also master theses) are planned based on the results of this thesis and with respect to the author's role as lecturer at the University of Applied Sciences in Mannheim.

**Research Projects & Funding.** During the course of the thesis, the *SeDiCo* idea with its FVPD approach was funded by the above-mentioned institutions. Closely related to the results of the thesis and to the above-mentioned topics, further research projects in cooperation with partners from industries and other research groups in national as well as international contexts should be acquired. The results of the thesis build an excellent foundation for additional project proposals in order to further pursue the FVPD idea of *SeDiCo* with funded projects.

**Invited Talks.** Furthermore, the author was invited by several institutions to talk and present new ideas about Cloud Security based on Partitioned and Distributed Databases. These talks are listed as follows:

1. 10/2016:

   SeDiCo - Query Optimization in Vertically Distributed Databases. Mannheimer Informatik-Kolloquium. Mannheim, Germany 2016. (in German).

2. 12/2014:

   SeDiCo - Towards a Framework for a Secure and Distributed Cloud Data Store. Mannheimer Informatik-Kolloquium. Mannheim, Germany 2014. (in German).

3. 07/2014:

   SeDiCo - Towards a Framework for a Secure and Distributed Cloud Data Store. German Chamber of Industry and Commerce Frankfurt. Frankfurt a. M., Germany 2014. (in German).

4. 01/2014:

   Business Process Modeling Repository. Foundations and Challenges of Change in Ontologies and Databases. University Bolzano, Italy 2014.

5. 07/2013:

   How to Handle Complex Data with Distributed Data Systems in the Cloud. Big Data Conference. Mannheim, Germany 2013. (in German).

6. 01/2013:

   Vertical Database Partitioning in the Cloud. Commit-Workshop Datenmanagement. University of Applied Sciences, Mannheim 2013. (in German).

7. 05/2012:

   Towards a Framework for a Distributed and Secure Cloud Datastore. WiWiTa 2012. Wismarer Wirtschaftsinformatiktage. Wismar 2012. (in German).

The dissemination of this thesis might generate additional attention to *SeDiCo* and its FVPD methodology. Hence, further invited talks might also become possible in the near future.

## Key Scientific and Applied Scientific Contributions

With respect to the tasks defined in the introduction of the thesis, it can be concluded that all tasks have successfully been accomplished. Namely, theses tasks were defined as listed in Table 8.7.

The results and the contribution to the current state-of-the-art, as defined at the beginning of this thesis, can be concluded as follows:

**Contribution 1:** Definition of a *Security-by-Distribution* Principle for Relational Databases

This thesis picked up the *Security-by-Distribution* approach and formalized it to substantiate it with the relational calculus to have a proper formal theoretical background. This was then used to prove the correctness of the principle and of all developed query mechanisms. After that, the evaluation results were used to formulate the research problem and to develop the

Table 8.7: Successfully Conducted Tasks

| Number | Task | Ref. to Thesis Chapter |
|---|---|---|
| 1 | The definition of a methodology for creating an FVPD schema for relational data and a proof of the correctness of the methodology | 1 |
| 2 | The conceptualization of adequate query mechanisms for relational FVPD data sets | 4 |
| 3 | The implementation of these relational query mechanisms in Java | 5 |
| 4 | The evaluation of these relational query mechanisms in terms of their response time | 6 |
| 5 | The comparison of all developed relational query mechanisms against each other and against the initial *SeDiCo* implementation | 7 |
| 6 | The application of the FVPD methodology in the Semantic Web with Resource Description Framework-based (RDF-based) data | 8 |

thesis' hypotheses. The TPC-W benchmark used in these works was also used to evaluate the query mechanisms in the presented thesis. With the verification of all hypotheses and the formal proofs, this thesis was able to prove that the developed query mechanisms indeed are able to enhance the response time.

**Contribution 2:** Development of Vertical Query Mechanisms

Due to the above-mentioned performance degrades while querying the vertically distributed data, this thesis developed 3 query strategies to tackle these issues. This thesis also formalized the strategies and with this, substantiated them with a formal background (i.e. the relational calculus). Hence, the correctness of all approaches could be proved and their complexity could be analyzed.

**Contribution 3:** Integration of Query Mechanisms into the *SeDiCo* Framework

Based on the above-mentioned results, the author was able to integrate them into the overall *SeDiCo* framework, such that it can be applied in a unique and transparent way.

This thesis developed a detailed evaluation with a detailed comparison and a discussion of all query mechanisms.

**Contribution 4:** Response Time Evaluation and Query Mechanism Classification

With respect to the before-mentioned results, this thesis picked up all evaluations and extended them with a detailed overview and interpreted and concluded the evaluation figures. The query mechanisms could be classified and concrete recommendations can now be given which mechanism is suitable in which use case scenario or for which database workload.

**Contribution 5:** Transfer of the *SeDiCo* Approach to other Databases

The Semantic Web as an interesting application domain could be identified during the course of this thesis. Here, the adoption of the *Security-by-Distribution* approach to RDF-based data sets which are queried with SPARQL were challenging issues. Accordingly, this thesis conducted a deep analysis of the theoretical background (i.e. relational calculus which is also used for SPARQL as it is closely related to SQL), added a formal correctness proof, and considered its complexity. Hence, it could be proved that the *Security-by-Distribution* approach is also applicable in other application domains.

**Contribution 6:** Further (Indirect) Related Work

This thesis subsumed the challenging topics *Cloud Computing*, *Security-by-Distribution*, *Performance*, *Scalability*, and *Reliability* under the *SeDiCo* umbrella and adapted the discussed issues to the focus of this thesis.

All in all, the author (and the respective co-authors) were able to publish a total of 16 research papers; the author was able to supervise a total of 11 student works (i.e. bachelor theses and seminar works), and there were 7 invited talks given by the author.

This leads to the outlook where the attention to future work tasks and interesting further research work is drawn.

# Outlook

Response time, besides data security and privacy, is a key issue in todays applications either they are based on physical hardware servers or on virtualized cloud infrastructures. The evaluation showed that there are viable approaches to achieve an adequate response times. This pushes the *SeDiCo* framework towards more practical usage scenarios, which involves further research topics. These topics

are now outlined to provide an overview about the future development of the framework.

The following topics emerged during the work on the *SeDiCo* framework and during the work on this thesis. Unfortunately, these topics cannot be dealt with here in order to not exceeding the scope of this thesis. However, they are mentioned here as future work topics to give an overview about further challenging tasks for interested framework developers, researchers, students, doctoral candidates, etc.

## Future *SeDiCo* Development

The future development of the *SeDiCo* framework concentrates primarily on the issues described in this section. These challenges are ideal to be addressed in seminar papers, student research projects or even bachelor or master theses.

**Primary Key Challenges.** In its current implementation it is mandatory that every relation that should be divided according to the FVPD principle, has a single primary key defined. This has two implications besides the fact that relations without a primary key cannot be partitioned and distributed: firstly, the data type of the defined primary key has a great influence on the join algorithms developed in the *query rewriting* approach. Secondly, relations with a compound primary key may cause erroneous behavior, as it is not further specifiable which of the primary key columns should be used for the join of the FVPD data. Here, a possible solution would be the creation of an artificial primary key on top of the compound primary key. Furthermore, functional dependencies of attributes to (compound) primary keys have to be taken into concern in this challenge.

Another issue with respect to this is the preservation of database constraints, i.e. foreign keys, unique constraints, etc. At the moment, these constraints are not considered in the FVPD approach. Here, the application logic (e.g. the ORM) has to enforce these constraints. The support of these constraints would require to search the relations meta data and to transfer them to the FVPD partitions accordingly. Yet, this is considered as a future work task as currently the ORM is capable of mapping the constraints to the partitions.

**Reporting in Distributed Databases.** Database reporting (e.g. query, update or other workload statistics) achieves a complexity similar to distributed

database. Here, an advanced reporting mechanism that integrates various different database systems and analyzes the workload against the FVPD partitions would be required.

**Partitioning and Distributing at Runtime.** Avoiding the fixed vertical partitioning and distribution scheme is another aim. This would allow to moving partitions between databases and clouds at runtime. Moreover, a repartitioning of the relations on demand at runtime would mean more flexibility and would raise the level of security.

**Row-based Security.** Another contribution to an increased level of security would be the integration of row-based security features as current database versions e.g. (PostgresSQL, 2016) do. This means that user privileges (permissions to read, write, etc.) can be defined for every user and for every single row specifically. Here, the practical usage of this feature in the near future will show if this is a viable approach and whether it should also be supported by *SeDiCo* or not.

**Mobile Platform Integration.** The adoption of the *SeDiCo* framework to other (especially mobile platforms) is thinkable. This would mean the transformation of the *SeDiCo* client, which is currently developed in Java, to other programming languages such as C#, Python, Swing or other prominent ones (TIOBE, 2016), or even on mobile platforms such as Android, iOS or Windows-Phone. Yet, in Cloud Computing environments where long lasting operations are performed in the cloud and only the results are delivered to the clients, this is an interesting approach.

**CRUD Performance Evaluation.** Further performance measurements with respect to the create, update, and delete, i.e. not only queries, but also the performance of data manipulation statements would be useful.

**Data Encryption.** The integration of encryption mechanisms, e.g. proposed in (Huber et al., 2013) is another interesting part, but is due to the already existing performance challenges, not in the focus of this work. Encrypting data would mean additional overhead and an additional performance decrease. The goal of this thesis is to make this approach comparable to a non-partitioned

and non-distributed architecture and then consecutively to integrate advanced encryption mechanisms (e.g. a public key infrastructure). Here, (Achenbach et al., 2011), (Huber et al., 2013) with Cumulus4j and (Popa et al., 2011) with CryptDB address this challenge and build a good foundation for FVPD data sets, concerning the encryption and the corresponding key management.

## Future Research Work

This section outlines further issues in a broader context of *SeDiCo*, which raise more complex and challenging tasks that require further research works in form of research papers, national or even international research projects or PhD theses.

**Data Classification.** Challenges that have come up in *SeDiCo* are the integration of the framework into heterogeneous enterprise infrastructures. Here, interesting questions such as the consolidation of existing data or suitable distribution approaches that differentiate between critical and less critical data emerge. Further literature dealing with these topics are (Chen & Liu, 2005) (Sood, 2012). Especially (Verykios et al., 2004) provide a good overview about techniques, that help to identify privacy-aware data in data mining. Therefore, transferring these approaches to the data classification problem of *SeDiCo* might be a promising way.

**Data Partitioning.** (Agrawal et al., 2004) provide a good overview about horizontal and vertical partitioning approaches. Analogous to this work, horizontal partitioning is considered as the row-based distribution of rows according to a certain value or criteria, whereas vertical partitioning is the column-based distribution of tuples. Both approaches are used to improve the overall database performance separately (vertical approaches e.g. (Rodriguez & Li, 2011), horizontal approaches e.g. (Huang et al., 2013), but also simultaneously e.g. (Alsultanny, 2010) (Agrawal et al., 2004). The usage of both approaches at the same time is considered a feasible way to improve the access of the data. This is referred to as diagonal partitioning. The future ongoing work will show, if this might even be a strategy for *SeDiCo*. Another challenge is to distribute the partitions efficiently across available computing nodes (i.e. distributed databases in different clouds) to minimize the data access times. This is closely related to NoSQL systems and the corresponding horizontal distribution of nodes and data sets. A well-documented

approach for this horizontal data distribution, including encryption of data was developed by (Neves et al., 2013).

**NoSQL.** NoSQL databases are an interesting topic for the future work but out of the scope of this thesis. Moreover, there is currently a great heterogeneity of NoSQL vendors and architectures (NoSQL Archive, 2016) and it is not predictable at the moment which kinds of databases and vendors will gain remarkable market shares[21]. So in order to achieve valuable and reusable results, a certain market consolidation is waited for.

[21]Currently, there are some trends, e.g. SAP HANA (SAP, 2016) as a column store, or MongoDB (MongoDB, 2016) as a document store, but especially in the Open Source community, no outstanding architecture or system like (MySQL in relational databases) can be determined.

# Declaration of Originality

Hereby, I declare that I have composed the presented thesis independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This work has neither been previously submitted to another authority nor has it been published yet.

Place, Date, Signature

(*Jens Kohler*)

# Acknowledgments

Although only my name appears on the cover of this dissertation, I would like to take the opportunity of expressing my sincere gratitude to my advisor Prof. Dr. Kiril Simov. I have been amazingly fortunate to have an advisor who gave me the freedom to explore and elaborate the key concepts of this thesis on my own, and at the same time lead me as a guide to relevant research topics and theoretical backgrounds.

I am also very grateful to Prof. Dr. Thomas Specht who has been always there to listen and to give advice. I am deeply thankful for the long discussions that helped me sort out the concepts and architectures of my work. I would also like to express my sincerest thanks to Prof. Dr. Sc. Galia Angelova for her guidance and her help to maintain the focus of this work.

I am also indebted to all the members of the Department of Computer Science with whom I have interacted during my research. Thank you - Miriam, Georg, Thomas, Michael, Thomas, Thorsten and Michael - for giving me the freedom to follow my ideas, for the welcome distractions and for the mind-opening discussions. My closest friends - Britta, Björn, Marleen, Felix, Sarah, and Christopher - have helped me to stay sane through these difficult years. Their support and care helped me to overcome setbacks and stay focused on my work. I greatly value their friendship and I deeply appreciate their belief in me. Most importantly, none of this would have been possible without the love and patience of my entire family; thank you, mom and dad for creating the environment and the cornerstones for my career and for your infinite love and support. I would also like to express my heart-felt gratitude to my extended family Margret and Martin for raising the most beautiful, love-filled and thoughtful wife - Myriam - that I could ever imagine. Finally, I envy my children - Emilia and Sofia - for the curiosity and eagerness to learn new things and I am grateful for showing me the real meaning of life.

# References


Achenbach, D., Gabel, M., & Huber, M. (2011). MimoSecco: A Middleware for Secure Cloud Storage. In *Proceedings of the 18th ispe international conference on concurrent engineering* (pp. 175–181). Boston, USA: Springer London.

Aggarwal, G., Bawa, M., Ganesan, P., Garcia-Molina, H., Kenthapadi, K., Motwani, R., ... Xu, Y. (2005). Two can keep a secret: A distributed architecture for secure database services. *Cidr 2005*, 186–199. Retrieved from `http://ilpubs.stanford.edu:8090/659/` doi: 10.1.1.86.9485

Agrawal, S., Narasayya, V., & Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 acm sigmod international conference on management of data* (pp. 359–371). New York, USA. doi: 10.1145/1007568.1007609

Akioka, S., & Muraoka, Y. (2010). HPC Benchmarks on Amazon EC2. In *2010 ieee 24th international conference on advanced information networking and applications workshops* (pp. 1029–1034). Perth, Australia. doi: 10.1109/WAINA.2010.166

Alsultanny, Y. (2010). Database management and partitioning to improve database processing performance. *Journal of Database Marketing and Customer Strategy Management*, *17*(3), 271–276. doi: 10.1057/dbm.2010.14

Amazon. (2016). *Amazon EC2 Website.* Retrieved 2016-02-01, from `https://aws.amazon.com/ec2/?nc1=h_ls`

Apache. (2015). *Apache Jena Website: SDB Documentation.* Retrieved 2016-02-01, from `https://jena.apache.org/documentation/sdb/`

Apache. (2016a). *Apache CloudStack.* Retrieved 2016-02-01, from `https://cloudstack.apache.org/`

Apache. (2016b). *Apache jclouds Website.* Retrieved 2016-02-01, from `https://jclouds.apache.org/reference/providers/`

Apache. (2016c). *Apache Jena Website.* Retrieved 2016-02-01, from `https://`

jena.apache.org/index.html

Apache. (2016d). *DeltaCloud Website.* Retrieved 2016-02-01, from https://deltacloud.apache.org

Apache. (2016e). *LibCloud Website.* Retrieved 2016-02-01, from https://libcloud.apache.org

Arias, M., Fernández, J. D., Martínez-Prieto, M. A., & de la Fuente, P. (2011). An Empirical Study of Real-World SPARQL Queries. *arXiv preprint arXiv: 1103.5043*, 10–13. Retrieved from http://arxiv.org/abs/1103.5043 doi: 10.1016/j.postharvbio.2004.03.006

Arora, I., & Gupta, A. (2012). Cloud Databases: A Paradigm Shift in Databases. *International Journal of Computer Science*, *9*(4), 77–83.

Ayani, R., Teo, Y. M., & Chen, P. (2002). Cost-based proxy caching. In *Proceedings of international symposium on distributed computing & applications to business, engineering & science* (pp. 218–222). Wuxi, China.

Bauer, C., King, G., & Gregory, G. (2007). *Java Persistence with Hibernate.* Greenwich, Connecticut: Manning Publications Company.

Berners-Lee, T. (2006). *Linked Data Principles.*

Berners-Lee, T. (2009). *W3C Website: Linked Data.* Retrieved 2016-02-01, from https://www.w3.org/DesignIssues/LinkedData.html

Bertino, E., & Sandhu, R. (2005). Database security - concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, *2*(1), 2–19. doi: 10.1109/TDSC.2005.9

Betz, H., Hose, K., & Sattler, K. (2012). Learning from the History of Distributed Query Processing. In *Third international workshop on consuming linked data (cold2012)* (Vol. 905, pp. 15–26). Boston, USA: CEUR-WS.

Binz, T., Breiter, G., Leyman, F., & Spatzier, T. (2012). Portable cloud services using TOSCA. *IEEE Internet Computing*, *16*(3), 80–85. doi: 10.1109/MIC.2012.43

Bizer, C., & Schultz, A. (2001). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, *5*(2), 1–24. doi: 10.4018/jswis.2009040101

Bohn, R. B., Messina, J., Liu, F., Tong, J., & Mao, J. (2011). NIST cloud computing reference architecture. In *Proceedings - 2011 ieee world congress on services, services 2011* (pp. 594–596). Gaithersburg, MD.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2Nd Edition).*

Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., & Reinwald, B. (2004).

Adaptive Database Caching with DBCache. *IEEE Data Engineering Bulletin*, *27*(2), 11–18.

Buil-Aranda, C., Polleres, A., & Umbrich, J. (2014). Strategies for Executing Federated Queries in SPARQL1.1. *The Semantic Web ISWC 2014 (Lecture Notes in Computer Science)*, *8797*, 390–405. doi: 10.1007/978-3-319-11915-1_25

Carey, M. J., DeWitt, D. J., & Naughton, J. F. (1993). The 007 Benchmark. In ACM New York (Ed.), *Proceedings of the 1993 acm sigmod international conference on management of data* (Vol. 22, pp. 12–21). NY, USA. doi: 10.1145/170036.170041

Carlton, D. (2013). *Cloud Computing 2014: Ready for Real Business?* Retrieved 2014-08-11, from `http://www.mscmalaysia.my/sites/all/themes/mscmalaysia/images/cloud\-page/cloud\-pdf/Morning\_2\_Gartner\_DarrylCarlton.pdf`

Cattell, R. G. (1994). *Object Data Management: Object-Oriented and Extended*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., & Zwaenepoel, W. (2003). Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *Proceedings of the acm/ifip/usenix 2003 international conference on middleware* (pp. 242–261). Rio de Janeiro, Brazil. doi: 10.1007/3-540-44892-6_13

Chen, K., & Liu, L. (2005). Privacy Preserving Data Classification with Rotation Perturbation. In *Fifth ieee international conference on data mining (icdm'05)* (pp. 589–592). Housten, Texas, USA. doi: 10.1109/ICDM.2005.121

Cloud Security Alliance. (2011). *Security Guidance for Critical Areas of Focus in Cloud Computing V3.0.* Retrieved 2016-02-01, from `https://cloudsecurityalliance.org/download/security-guidance-for-critical-areas-of-focus-in-cloud-computing-v3/`

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377–387. doi: 10.1145/362384.362685

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st acm symposium on cloud computing - socc '10* (pp. 143–155). Indianapolis, USA: ACM Press. doi: 10.1145/1807128.1807152

Cudre-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., ... Wylot, M. (2013). NoSQL databases for RDF: An empirical evaluation. In *Lecture notes in computer science (including subseries lecture*

*notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 8219 LNCS, pp. 310–325). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-41338-4_20

Cyganiak, R. (2016). *D2RQ Website.* Retrieved 2016-02-01, from `http://d2rq.org/`

Cyganiak, R., & Cyganiak, R. (2005). *Technical Report: A relational algebra for SPARQL* (Vol. 12; Tech. Rep.). Bristol: Hewlett-Packard Development Company. Retrieved from `http://fog.hpl.external.hp.com/techreports/2005/HPL-2005-170.pdf`

Darari, F., Nutt, W., Pirrò, G., & Razniewski, S. (2013). Completeness statements about RDF data sources and their use for query answering. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 8218 LNCS, pp. 66–83). doi: 10.1007/978-3-642-41335-3_5

Darari, F., Razniewski, S., & Nutt, W. (2014). Bridging the Semantic Gap Between RDF and SPARQL Using Completeness Statements. In *Proceedings of the 2014 international conference on posters & demonstrations track - volume 1272* (pp. 269–272). Aachen, Germany, Germany: CEUR-WS.org.

Das, S., Sundara, S., & Cyganiak, R. (2012). *W3C Website: R2RML - RDB to RDF Mapping Language.* Retrieved 2016-02-01, from `https://www.w3.org/TR/r2rml/`

Davision, B. D. (2001). A Web caching primer. *IEEE Internet Computing*, *5*(4), 38–45. doi: 10.1109/4236.939449

DBLP. (2016). *DBLP Website.* Retrieved 2016-02-01, from `http://dblp.uni-trier.de/`

DBPedia. (2016). *DBPedia Data Set.* Retrieved 2016-02-01, from `http://wiki.dbpedia.org/services-resources/datasets/dbpedia-datasets`

DIN ISO 27000. (2011). *DIN ISO/IEC 27000.* Retrieved 2016-02-01, from `https://www.beuth.de/de/norm-entwurf/din-iso-iec-27000/243433889`

Dongarra, J. J. (1990). The LINPACK benchmark: An explanation. In (pp. 456–474). London, UK: Chapman & Hall, Ltd.

Duan, S., Kementsietsidis, A., Srinivas, K., & Udrea, O. (2011). Apples and Oranges : A Comparison of RDF Benchmarks and Real RDF Datasets. In *Sigmod '11 proceedings of the 2011 acm sigmod international conference on management of data* (pp. 145–155). Athens, Greece: ACM. doi: 10.1145/1989323.1989340

Duerst, M., & Suignard, M. (2005). *IETF IRI, RFC 3987.* Retrieved 2016-02-01, from https://www.ietf.org/rfc/rfc3987.txt

Elmasri, R., & Navathe, S. B. (2015). *Fundamentals of Database Systems* (7th editio ed.). London, UK: Pearson Education, UK. doi: 10.1016/S0026-2692(97)80960-3

European Commission. (1995). *Directive 95/46/EC.* Retrieved from http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML

Fan, B., Andersen, D., & Kaminsky, M. (2013). MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Nsdi'13 proceedings of the 10th usenix conference on networked systems design and implementation* (pp. 371–385). Lombard, Ilinois, USA.

FluidOperations. (2016). *Fluid Operations Website: FedX.* Retrieved 2016-02-01, from https://www.fluidops.com/en/company/training/open_source

Franklin, M. J., Carey, M. J., & Livny, M. (1997). Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, *22*(1), 315–363. doi: 10.1145/261124.261125

Franz, T., Schultz, A., Sizov, S., & Staab, S. (2009). TripleRank: Ranking Semantic Web data by tensor decomposition. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 5823 LNCS, pp. 213–228). doi: 10.1007/978-3-642-04930-9_14

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd edition ed.). New York, USA: Prentice Hall. doi: 10.1145/253262.253287

Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., & Tomasic, A. (2008). Scalable query result caching for web applications. *Proceedings of the VLDB Endowment*, *1*(1), 550–561. doi: 10.14778/1453856.1453917

Gartner. (2013). *Gartner's 2013 Hype Cycle for Emerging Technologies Maps Out Evolving Relationship Between Humans and Machines.* Retrieved 2016-02-01, from http://www.gartner.com/newsroom/id/2575515

Gens, F., & Shirer, M. (2013). *IDC Forecasts Worldwide Public IT Cloud Services Spending to Reach Nearly $108 Billion by 2017 as Focus Shifts from Savings to Innovation.* Retrieved 2016-02-01, from http://www.idc.com/getdoc.jsp?containerId=prUS24298013

GeoNames. (2016). *GeoNames Website.* Retrieved 2016-02-01, from http://

www.geonames.org/

Ghandeharizadeh, S., & Mutha, A. (2014). An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions. In *Proceedings of the 16th international conference on information integration and web-based applications & services - iiwas '14* (pp. 64–70). New York, USA: ACM Press. doi: 10.1145/2684200.2684285

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, *33*(2), 51–59. doi: 10.1145/564585.564601

Gimenez-Garcia, J. M., Fernandez, J. D., & Martinez-Prieto, M. (2014). MapReduce-based Solutions for Scalable SPARQL Querying. *Open Journal of Semantic Web (OJSW)*, *1*(1), 1–18. Retrieved from http://dataweb.infor.uva.es/wp-content/uploads/2014/03/ojsw14.pdf

GitHub. (2016). *GitHub Website.* Retrieved 2016-02-01, from https://github.com/

Görlitz, O., & Staab, S. (2011). Federated Data Management and Query Optimization for Linked Open Data. *New Directions in Web Data Management (Studies in Computational Intelligence)*, *331*, 109–137. doi: 10.1007/978-3-642-17551-0_5

Graefe, G. (2011). New algorithms for join and grouping operations. *Computer Science - Research and Development*, *27*(1), 3–27. doi: 10.1007/s00450-011-0186-9

Grund, M., Cudre-Mauroux, P., & Madden, S. (2011). A Demonstration of HYRISE A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, *4*(12), 1434–1437.

Grund, M., Schaffner, J., Krueger, J., Brunnert, J., & Zeier, A. (2010). The Effects of Virtualization on Main Memory Systems. In *Proceedings of the sixth international workshop on data management on new hardware* (pp. 41–46). New York, NY, USA: ACM. doi: 10.1145/1869389.1869395

Guo, Y., Pan, Z., & Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, *3*(2-3), 158–182. doi: 10.1016/j.websem.2005.06.005

Haase, P., Mathäß, T., & Ziller, M. (2010). An evaluation of approaches to federated query processing over linked data. In *I-semantics '10 proceedings of the 6th international conference on semantic systems* (Vol. 5, pp. 1–9). Graz, Austria: ACM. doi: 10.1145/1839707.1839713

Harris, S., Lamb, N., & Shadbolt, N. (2009). 4store: The Design and Implemen-

tation of a Clustered RDF Store. In *Scalable semantic web knowledge base systems - ssws2009* (pp. 81–96). Washington DC, USA: CEUR-WS.

Harris, S., & Seaborne, A. (2013). *W3C Website: SPARQL 1.1 Query Language.* Retrieved 2016-02-01, from `http://www.w3.org/TR/sparql11-query/`

Harth, A., Umbrich, J., Hogan, A., & Decker, S. (2007). YARS2: a federated repository for querying graph structured data from the web. *Lecture Notes in Computer Science*, *4825*, 211–224.

Hevner, A., & Chatterjee, S. (2010). *Design Research in Information Systems* (Vol. 22). Boston, MA: Springer US. doi: 10.1007/978-1-4419-5653-8

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. doi: 10.2307/25148625

Hewlett Packard. (2016). *Helios Eucalyptus Website.* Retrieved 2016-02-01, from `http://www8.hp.com/us/en/cloud/helion-overview.html`

Hofmann, M., & Beaumont, L. (2005). *Content networking: architecture, protocols, and practice*. Burlington, Massachusetts, USA: Morgan Kaufman Publ Inc.

Hofmann, M., Feig, E., & Zhang, J. (2009). *From SaaS to XaaS: Evolution and Outlook of Software Cloud.* Retrieved 2016-02-01, from `http://www.thecloudcomputing.org/2009/1/panels.html#Panel2`

Huang, C., Hu, W., Shih, C., Lin, B., & Cheng, C. (2013). The improvement of auto-scaling mechanism for distributed database - A case study for MongoDB. In *15th asia-pacific network operations and management symposium (apnoms), 2013* (pp. 1–3). Hiroshima, Japan.

Huber, M., Gabel, M., Schulze, M., & Bieber, A. (2013). Cumulus4j: A Provably Secure Database Abstraction Layer. In *Proceedings of cd-ares 2013 workshops: Mocrysen and secihd* (pp. 180–193). Regensburg, Germany.

Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009). A classification of object-relational impedance mismatch. In *Proceedings - 2009 1st international conference on advances in databases, knowledge and data applications, dbkda 2009* (pp. 36–43). Gosier, Guadeloup. doi: 10.1109/DBKDA.2009.11

ISO/IEC. (2005). *ISO/IEC 19501:2005 - Information technology - Open Distributed Processing - Unified Modeling Language (UML).* Retrieved 2017-07-26, from `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=32620`

ISO/IEC. (2011). *Database Languages -SQL- Part 1 - 14. Standardization Document: ISO/IEC 9075:2011* (Tech. Rep.). Geneva, Switzerland: Author.

Kaiser, L. (2013). *Evaluation of Various Cloud Abstraction Layers for a Distributed Cloud Datastore.* Bachelorthesis at Institute for Enterprise Computing,

University of Applied Sciences Mannheim, Mannheim Germany.

Kanehisa Laboratories. (2016). *KEGG: Kyoto Encyclopedia of Genes and Genomes.* Retrieved 2016-02-01, from `http://www.genome.jp/kegg/`

Kohler, J., Simov, K., Fiech, A., & Specht, T. (2015). On The Performance Of Query Rewriting In Vertically Distributed Cloud Databases. In *Proceedings of the international conference advanced computing for innovation acomin 2015.* Sofia, Bulgaria.

Kohler, J., Simov, K., & Specht, T. (2015). Analysis of the Join Performance in Vertically Distributed Cloud Databases. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, *6*(2). doi: 10.4018/IJARAS

Kohler, J., & Specht, T. (2012). SeDiCo - Towards a Framework for a Secure and Distributed Datastore in the Cloud. In *Proceedings of chip-to-cloud security forum 2012.* Nice, France.

Kohler, J., & Specht, T. (2014a). Ein Marktplatz für die Cloud: Vergleichbarkeit von Datenspeichern durch QoS-/SLA-Mapping. *Technologien für digitale Innovationen*, *1*(1).

Kohler, J., & Specht, T. (2014b). Vertical Query-Join Benchmark in a Cloud Database Environment. In *Proceedings of the 2nd ieee world conference on complex systems.* Agadir, Marocco.

Kohler, J., & Specht, T. (2014c). Vertical Update-Join Benchmark in a Cloud Database Environment. In *Proceedings of WiWiTa 2014 Wismarer Wirtschaftsinformatiktage. Wismar, Germany* (pp. 159–175). Wismar, Germany.

Kohler, J., & Specht, T. (2015a). Analysis of Cache Implementations in a Vertically Distributed Cloud Data Store. In *Proceedings of the 3rd ieee world conference on complex system.* Marrakesh, Morocco.

Kohler, J., & Specht, T. (2015b). Performance Analysis of Vertically Partitioned Data in Clouds Through a Client-Based In-Memory Key-Value Store Cache. In *Proceedings of the 8th international conference on computational intelligence in security for information systems.* Burgos, Spain: Springer.

Kohler, J., & Specht, T. (2015c). A Performance Comparison Between Parallel And Lazy Fetching in Vertically Distributed Cloud Databases. In *International conference on cloud computing technologies and applications - cloudtech 2015.* Marrakesh, Morocco: IEEE Computer Society.

Krueger, J., Grund, M., Zeier, A., & Plattner, H. (2010). Enterprise application-specific data management. In *Proceedings - ieee international enterprise distributed object computing workshop, edoc* (pp. 131–140). Vitoria, Brazil.

Li, L., & Gruenwald, L. (2012). Autonomous database partitioning using data mining on single computers and cluster computers. In *Proceedings of the 16th international database engineering and applications sysmposium on - ideas '12* (pp. 32–41). New York, New York, USA: ACM Press. doi: 10.1145/2351476.2351481

Lorey, J., & Naumann, F. (2013). Detecting SPARQL query templates for data prefetching. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 7882 LNCS, pp. 124–139). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-38288-8-9

Luo, Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., & Naughton, J. F. (2002). Middle-tier database caching for e-business. In *Proceedings of the 2002 acm sigmod international conference on management of data* (pp. 600–611). Madison, Wisconsin, USA.

Lutteroth, C., & Weber, G. (2009). Database synchronization as a service. In *Proceedings - ieee international enterprise distributed object computing workshop, edoc 2009* (pp. 84–91). Auckland, New Zealand. doi: 10.1109/EDOCW.2009.5332009

Manola, F., Miller, E., & McBride, B. (2014). *RDF 1.1 Primer Website W3C Working Group Note 25 February 2014.* Retrieved 2016-02-01, from `http://www.w3.org/TR/rdf11-primer/`

Martin, M., Unbehauen, J., & Auer, S. (2010). Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *Proceedings of 7th extended semantic web conference (eswc 2010)* (pp. 304–318). Heraklion, Crete, Greece: ACM. Retrieved from `http://www.springerlink.com/content/764m684325739v67/` doi: doi:10.1007/978-3-642-13489-0_21

Mattsson, U. (2008). How to Prevent Internal and External Attacks on Data - Securing the Enterprise Data Flow Against Advanced Attacks. *SSRN Electronic Journal*, *1*(1). doi: 10.2139/ssrn.1144290

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing* (Tech. Rep. No. 800-145). Gaithersburg, MD: National Institute of Standards and Technology (NIST). Retrieved 2016-02-01, from `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`

Melnik, S., Adya, A., & Bernstein, P. a. (2008). Compiling mappings to bridge applications and databases. *ACM Transactions on Database Systems*, *33*(4), 1–50. doi: 10.1145/1412331.1412334

Microsoft. (2016). *Microsoft Linq Website.* Retrieved 2016-02-01, from `https://`

msdn.microsoft.com/de-de/library/bb397926.aspx

Mishra, P., & Eich, M. H. (1992). Join processing in relational databases. *ACM Computing Surveys*, *24*(1), 63–113. doi: 10.1145/128762.128764

MongoDB. (2016). *MongoDB Website.* Retrieved 2016-10-01, from https://www.mongodb.org/

Montoya, G., Vidal, M. E., Corcho, O., Ruckhaus, E., & Buil-Aranda, C. (2012). Benchmarking federated SPARQL query engines: Are existing testbeds enough? In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 7650 LNCS, pp. 313–324). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-35173-0-21

Morsey, M., Lehmann, J., Auer, S., & Ngonga Ngomo, A. C. (2011). DBpedia SPARQL benchmark - Performance assessment with real queries on real data. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 7031 LNCS, pp. 454–469). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-25073-6_29

MySQL. (2016). *MySQL 5.6 Reference Manual Including MySQL Cluster NDB 7.3 Reference Guide.* Author. Retrieved 2016-02-01, from http://dev.mysql.com/doc/

Neves, B. A., Correia, M. P., Bruno, Q., Fernando, A., & Paulo, S. (2013). DepSky: dependable and secure storage in a cloud-of-clouds. In *Acm transactions on storage (tos)* (Vol. 9, pp. 31–46). ACM. doi: 10.1145/2535929

Ning, X., Jin, H., & Wu, H. (2008). RSS: A framework enabling ranked search on the semantic web. *Information Processing and Management*, *44*(2), 893–909. doi: 10.1016/j.ipm.2007.03.005

NoSQL Archive. (2016). *NoSQL Archive Website.* Retrieved 2016-02-01, from http://nosql-databases.org/

OECD. (2013). *2013 OECD Privacy Guidelines.* Organisation for Economic Co-operation and Development. Retrieved 2016-02-01, from http://www.oecd.org/sti/ieconomy/oecdguidelinesontheprotectionofprivacy.htm

Olston, C., & Widom, J. (2002). Best-effort cache synchronization with source cooperation. In *Proceedings of the 2002 acm sigmod international conference on management of data - sigmod '02* (pp. 73–84). New York, USA: ACM Press. doi: 10.1145/564691.564701

Omg. (2011). UML Infrastructure Specification, v2.4.1. *Omg*(August), 34. Retrieved from http://www.omg.org/spec/UML/2.4.1/Infrastructure/

PDF/ doi: 10.1007/s002870050092

Ontop. (2016). *Ontop Website.* Retrieved 2016-02-01, from http://ontop.inf.unibz.it/

Ontotext. (2016). *GraphDB Website.* Retrieved 2016-02-01, from http://ontotext.com/products/graphdb/

Oracle. (2016). *Oracle® Database VLDB and Partitioning Guide 11g Release 1 (11.1).* Retrieved 2016-02-01, from http://docs.oracle.com/cd/B28359_01/server.111/b32024/title.htm

Ottinger, J., Guruzu, S., & Mak, G. (2015). *Hibernate Recipes.* Apress, New York.

Özsu, M. T., & Valduriez, P. (2011). *Principles of Distributed Database Systems* (3rd editio ed., Vol. 12). New York, USA: Springer. doi: 10.1007/978-1-4419-8834-8

Pearson, S. (2013). Privacy, Security and Trust in Cloud Computing. In *Privacy and security for cloud computing se - 1* (pp. 3–42). Springer London. doi: 10.1007/978-1-4471-4189-1_1

Pérez, J., Arenas, M., & Gutierrez, C. (2006). Semantics and Complexity of SPARQL. *The Semantic Web - ISWC 2006*, *4273*, 30–43. doi: 10.1007/11926078

Plattner, H. (2013). *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases.* Berlin Heidelberg, Germany: Springer.

Podlipnig, S., & Böszörmenyi, L. (2003). A survey of Web cache replacement strategies. *ACM Computing Surveys*, *35*(4), 374–398. doi: 10.1145/954339.954341

Popa, R. A., Redfield, C. M. S., Zeldovich, N., & Balakrishnan, H. (2011). CryptDB. In *Proceedings of the twenty-third acm symposium on operating systems principles - sosp '11* (p. 85). New York, USA: ACM Press. doi: 10.1145/2043556.2043566

Ports, D. R. K., Clements, A. T., Zhang, I., Madden, S., & Liskov, B. (2010). Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the ninth usenix symposium on operating systems design and implementation* (pp. 279–292). Vancouver, BC, Canada.

PostgresSQL. (2016). *PostgreSQL Documentation 9.5.* Retrieved 2016-02-01, from http://www.postgresql.org/

Pritchett, D. (2008). BASE: An ACID Alternative. *Queue*, *6*(3), 48–55. doi: 10.1145/1394127.1394128

Qiao, S., & Özsoyo, Z. M. (2015). RBench : Application-Specific RDF Bench-

marking. In *Proceedings of the 2015 acm sigmod international conference on management of data* (pp. 1825–1838). Melbourne, VIC, Australia: ACM. doi: 10.1145/2723372.2746479

Quilitz, B. (2006). *DARQ Website.* Retrieved 2016-02-01, from `http://darq.sourceforge.net/`

Quilitz, B., & Leser, U. (2008). Querying distributed RDF data sources with SPARQL. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 5021 LNCS, pp. 524–538). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-68234-9_39

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., & Hausenblas, M. (2013). Querying over Federated SPARQL Endpoints - A State of the Art Survey. *arXiv preprint arXiv:1306.1723*. Retrieved from `http://arxiv.org/abs/1306.1723`

RedHat. (2016). *ORM Hibernate Documentation.* Retrieved 2016-02-01, from `http://hibernate.org/orm/documentation/5.0/`

Rodríguez, L., & Li, X. (2011). A dynamic vertical partitioning approach for distributed database system. In *Conference proceedings - ieee international conference on systems, man and cybernetics* (pp. 1853–1858). doi: 10.1109/ICSMC.2011.6083941

Rodriguez, L., & Li, X. (2011). A dynamic vertical partitioning approach for distributed database system. In *Conference proceedings - ieee international conference on systems, man and cybernetics* (pp. 1853–1858). Anchorage, Alaska USA. doi: 10.1109/ICSMC.2011.6083941

Rodríguez-Muro, M., & Rezk, M. (2015). Efficient SPARQL-to-SQL with R2RML mappings. *Journal of Web Semantics*, *33*, 141–169. doi: 10.1016/j.websem.2015.03.001

Rohilla, S., & Mittal, P. (2013). Database Security: Threats and Challenges. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3*(5), 810–813.

Russell, C. (2008). Bridging the Object-Relational Divide. *ACM Queue*, *6*(3), 18–29. doi: 10.1145/1394127.1394139

SAP. (2016). *SAP HANA Website.* Retrieved 2016-02-02, from `https://hana.sap.com/abouthana.html`

Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., & Tran, T. (2011). FedBench: A benchmark suite for federated semantic data query processing. In *Lecture notes in computer science (including subseries lecture notes in*

*artificial intelligence and lecture notes in bioinformatics)* (Vol. 7031 LNCS, pp. 585–600). doi: 10.1007/978-3-642-25073-6_37

Schmidt, M., Hornung, T., Meier, M., Pinkel, C., & Lausen, G. (2009). SP2bench: A SPARQL performance benchmark. In *Ieee 25th international conference on data engineering, 2009. icde '09.* (pp. 222 – 233). Shanghai, China: IEEE. doi: 10.1007/978-3-642-04329-1_16

Schwarte, A., Haase, P., Schmidt, M., Hose, K., & Schenkel, R. (2012). An Experience Report of Large Scale Federations. *arXiv preprint arXiv:1210.5403*. Retrieved from `http://arxiv.org/abs/1210.5403`

Sequeda, J. F., & Miranker, D. P. (2013). Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, *22*, 19–39. doi: 10.1016/j.websem.2013.08.002

Sesame. (2015). *Sesame Website: Sesame SAIL API Documentation.* Retrieved 2016-02-01, from `http://rdf4j.org/sesame/2.7/apidocs/org/openrdf/sail/rdbms/RdbmsStore.html`

Sesame. (2016). *Sesame Website.* Retrieved 2016-02-01, from `http://rdf4j.org/`

Sion, R. (2007). Secure data outsourcing. In *Proceedings of the 33rd international conference on very large data bases* (Vol. 4, pp. 1431–1432). Vienna, Austria.

Sivasubramanian, S., Pierre, G., van Steen, M., & Alonso, G. (2007). Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, *11*(1), 60–66. doi: 10.1109/MIC.2007.3

Son, J. H., & Kim, M. H. (2004). An adaptable vertical partitioning method in distributed systems. *Journal of Systems and Software*, *73*(3), 551–561. doi: 10.1016/j.jss.2003.04.002

Sood, S. K. (2012). A combined approach to ensure data security in cloud computing. *Journal of Network and Computer Applications*, *35*(6), 1831–1838. doi: 10.1016/j.jnca.2012.07.007

SPC. (2013). *SPC Benchmark 1 Official Specification.* California, USA: Storage Performance Council. Retrieved 2016-02-01, from `http://www.storageperformance.org/specs/SPC-1_SPC-1E_v1.14.pdf`

SPEC. (2016). *SPEC Website.* Retrieved 2016-02-01, from `https://www.spec.org`

SQLAlchemy. (2016). *SQLAlchemy Website.* Retrieved 2016-02-01, from `http://www.sqlalchemy.org/`

Steve, P., & Ushar, T. (2011). *Securing the Cloud - Using Encryption and Key Management to Solve Today's Cloud Security Chal-*

*lenges.* Colorado Springs, CO: Storage Networking Industry Association (SINA). Retrieved 2016-02-01, from https://www.google.at/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiIsv3ipe_JAhVL1ywKHTWNAEcQFggmMAA&url=http://www.snia.org/sites/default/education/tutorials/2011/spring/security/PateTambay_Securing_the_Cloud_K

Sweeney, L. (2002). Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *10*(05), 571–588. doi: 10.1142/S021848850200165X

Systap. (2016). *Blazegraph Website.* Retrieved 2016-02-01, from https://wiki.blazegraph.com

TIOBE. (2016). *TIOBE Index.* Retrieved 2016-02-01, from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

TPC. (2003). *TPC Benchmark W (Web Commerce) Specification Version 2.0r.* Retrieved 2016-02-01, from http://www.tpc.org/tpcw/default.asp

TPC. (2014). *TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.0.* San Francisco, USA: Transaction Processing Performance Concil. Retrieved 2016-02-01, from http://www.tpc.org/tpch/default.asp

Van Zyl, P., Kourie, D. G., & Boake, A. (2006). Comparing the performance of object databases and ORM tools. In *Proceedings of the 2006 annual research conference of the south african institute of computer scientists and information technologists on it research in developing countries - saicsit '06* (pp. 1–11). New York, USA: ACM Press. doi: 10.1145/1216262.1216263

Van Zyl, P., Kourie, D. G., Coetzee, L., & Boake, A. (2009). The influence of optimisations on the performance of an object relational mapping tool. In *Proceedings of the 2009 annual research conference of the south african institute of computer scientists and information technologists on - saicsit '09* (pp. 150–159). Vaal River, South Africa. doi: 10.1145/1632149.1632169

Verykios, V. S., Bertino, E., Fovino, I. N., Provenza, L. P., Saygin, Y., & Theodoridis, Y. (2004). State-of-the-art in privacy preserving data mining. *ACM SIGMOD Record*, *33*(1), 50. doi: 10.1145/974121.974131

Virtuoso. (2016). *Virtuoso Website.* Retrieved 2016-02-01, from http://virtuoso.openlinksw.com/

W3C. (2007). *W3C Semantic Web Stack.* Retrieved 2016-02-01, from https://www.w3.org/2007/03/layerCake.png

W3C. (2016a). *W3C Website: RDF and OWL Browsers.* Retrieved 2016-02-01, from https://www.w3.org/2001/sw/wiki/Category:RDF_or_OWL_Browser

W3C. (2016b). *W3C Website: SPARQL Implementations.* Retrieved 2016-02-01, from https://www.w3.org/wiki/SparqlImplementations

Wood, D. (2014). *W3C Website: What's new in RDF 1.1.* Retrieved 2016-02-01, from https://www.w3.org/TR/rdf11-new/

Wu, B., Zhou, Y., Yuan, P., Jin, H., & Liu, L. (2014). SemStore: A Semantic-Preserving Distributed RDF Triple Store. In *Proceedings of the 23rd acm international conference on conference on information and knowledge management* (pp. 509–518). Shanghai, China: ACM. doi: 10.1145/2661829.2661876

Yao, S. B., & Hevner, A. R. (1984). *A Guide to Performance Evaluation of Database Systems.* Washington, D.C., USA: NBS Special Publication.

# Appendix A

# List of Tables

# Appendix B

# List of Figures

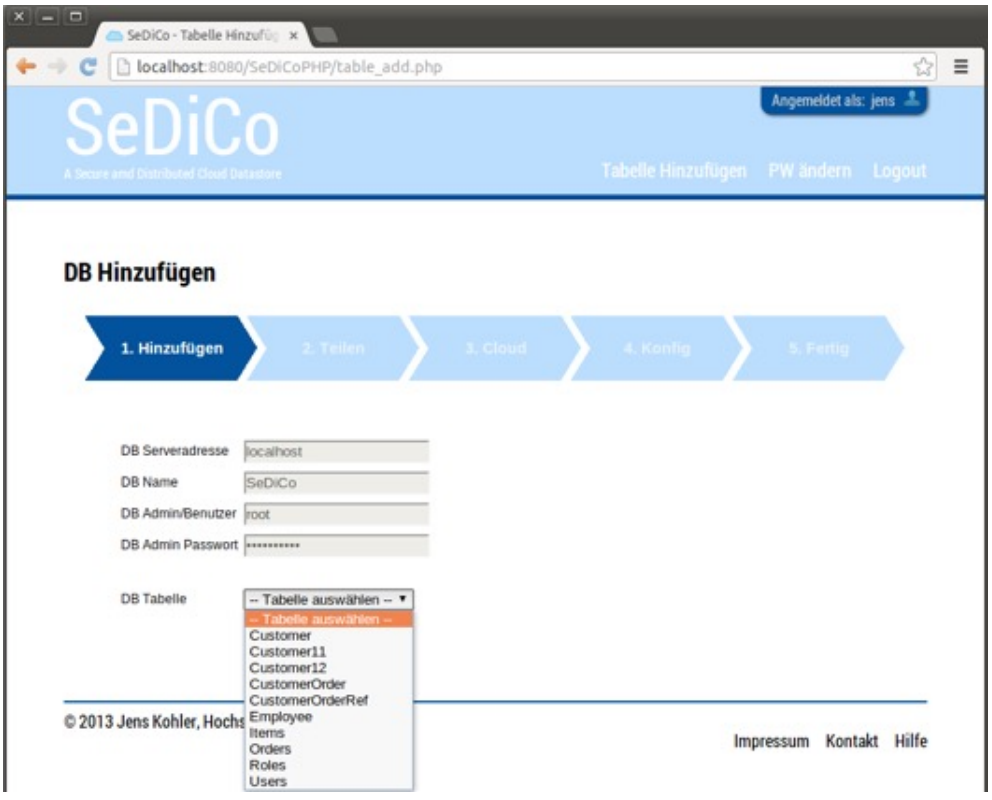
# Appendix C

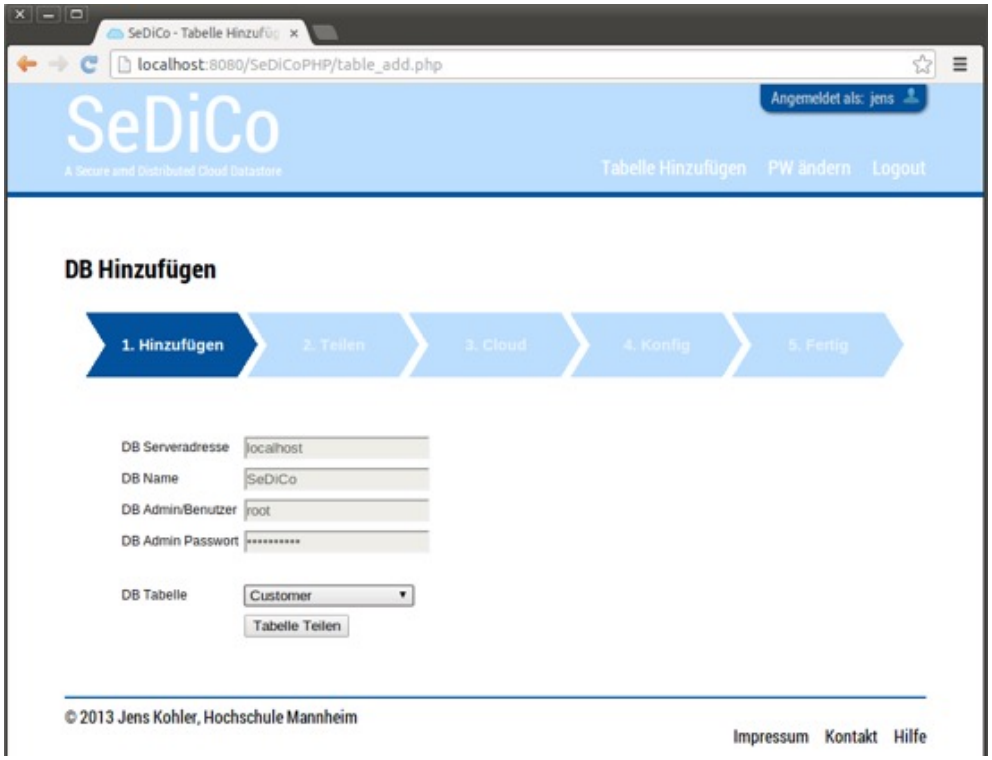
# Listings

# Appendix D

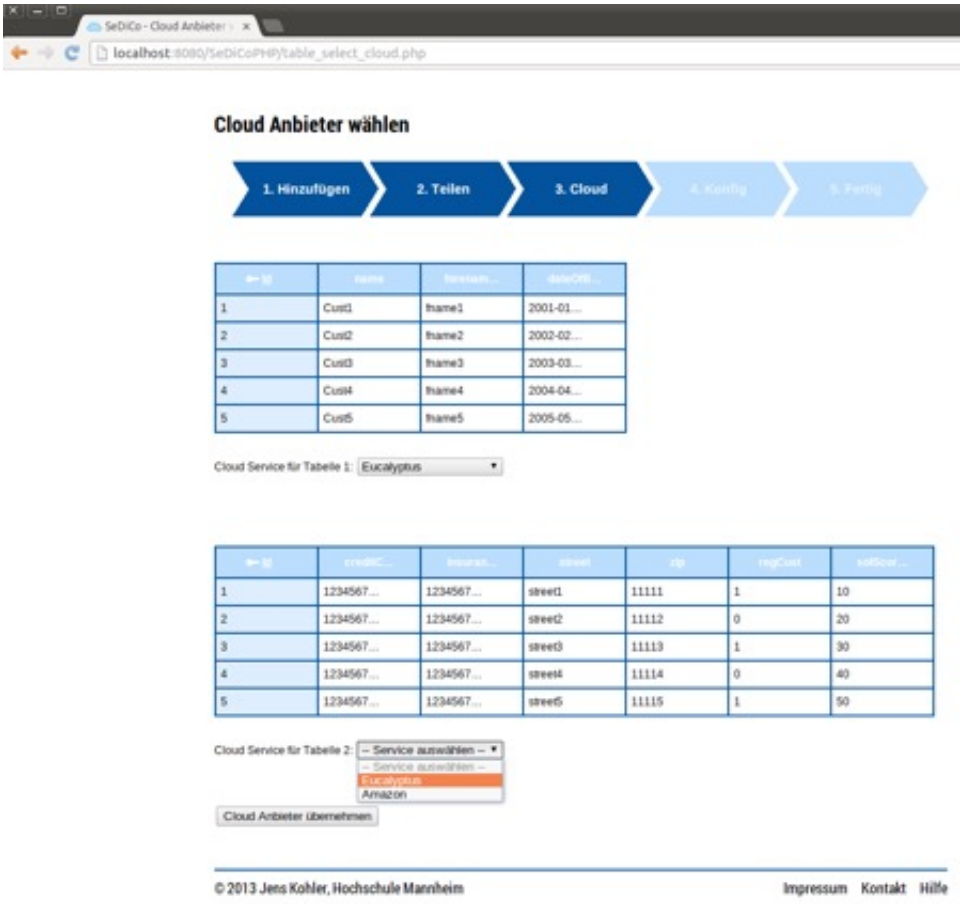
# *SeDiCo* Application Screenshots

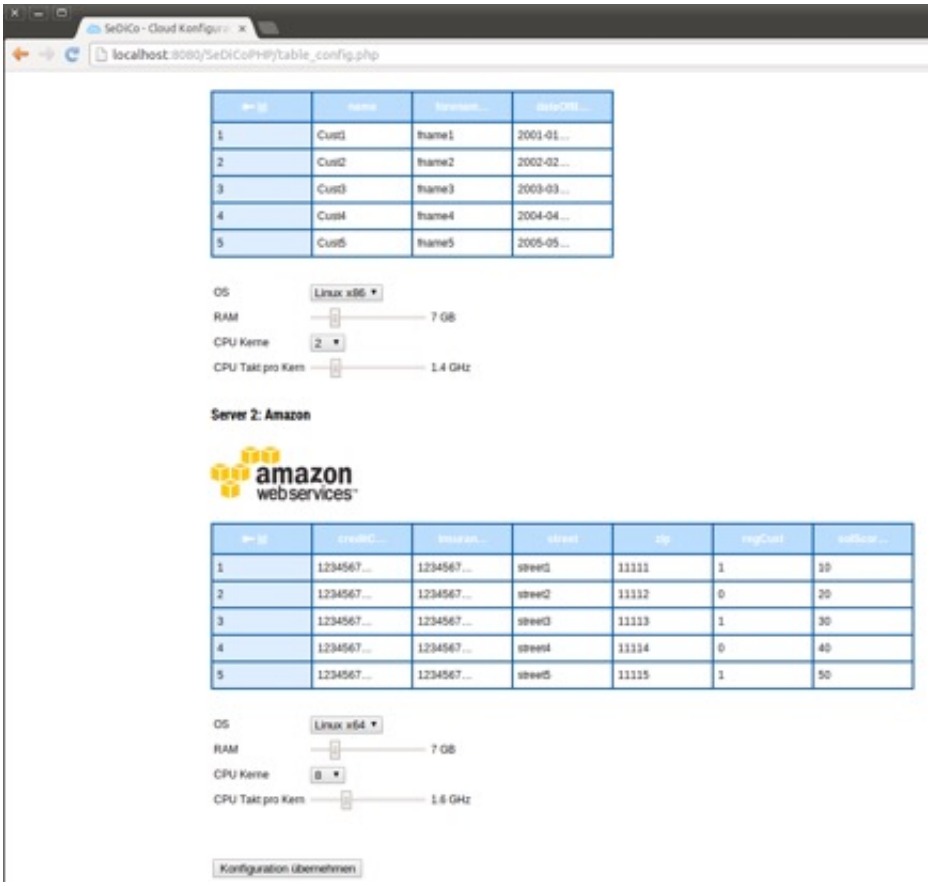

SeDiCo Startseite
localhost:8080/SeDiCoPHP/index.php
SeDiCo
A Secure and Distributed Cloud Datastore
Angemeldet als: jens
Tabelle Hinzufügen
PW ändern
Logout
Willkommen jens
Public Cloud
Hybrid Cloud
Private Cloud
SeDiCo
Datenbank
© 2013 Jens Kohler, Hochschule Mannheim
Impressum
Kontakt
Hilfe


SeDiCo - Tabelle Hinzufüg
localhost:8080/SeDiCoPHP/table_add.php
SeDiCo
A Secure and Distributed Cloud Datastore
Angemeldet als: jens
Tabelle Hinzufügen
PW ändern
Logout
DB Hinzufügen
1. Hinzufügen
2. Teilen
3. Cloud
4. Konfig
5. Fertig
DB Serveradresse
localhost
DB Name
SeDiCo
DB Admin/Benutzer
root
DB Admin Passwort
DB Tabelle
-- Tabelle auswählen --
-- Tabelle auswählen --
Customer
Customer11
Customer12
CustomerOrder
CustomerOrderRef
Employee
Items
Orders
Roles
Users
© 2013 Jens Kohler, Hochs
Impressum
Kontakt
Hilfe


SeDiCo - Tabelle Hinzufüg
localhost:8080/SeDiCoPHP/table_add.php
SeDiCo
A Secure and Distributed Cloud Datastore
Angemeldet als: jens
Tabelle Hinzufügen
PW ändern
Logout
DB Hinzufügen
1. Hinzufügen
2. Teilen
3. Cloud
4. Konfig
5. Fertig
DB Serveradresse
localhost
DB Name
SeDiCo
DB Admin/Benutzer
root
DB Admin Passwort
DB Tabelle
Customer
Tabelle Teilen
© 2013 Jens Kohler, Hochschule Mannheim
Impressum
Kontakt
Hilfe


SeDiCo - Tabelle trennen
localhost:8080/SeDiCoPHP/table_divide.php
SeDiCo
Angemeldet als: jens
Tabelle Hinzufügen
PW ändern
Logout
Tabelle teilen
1. Hinzufügen
2. Teilen
3. Cloud
4. Konfig
5. Fertig
Tabelle: Customer
Server: localhost
Tabelle teilen
© 2013 Jens Kohler, Hochschule Mannheim
Impressum
Kontakt
Hilfe


SeDiCo - Cloud Anbieter
localhost:8080/SeDiCoPHP/table_select_cloud.php
Cloud Anbieter wählen
1. Hinzufügen
2. Teilen
3. Cloud
4. Konfig
5. Fertig
1 Cust1 fname1 2001-01...
2 Cust2 fname2 2002-02...
3 Cust3 fname3 2003-03...
4 Cust4 fname4 2004-04...
5 Cust5 fname5 2005-05...
Cloud Service für Tabelle 1: Eucalyptus
1 1234567... 1234567... street1 11111 1 10
2 1234567... 1234567... street2 11112 0 20
3 1234567... 1234567... street3 11113 1 30
4 1234567... 1234567... street4 11114 0 40
5 1234567... 1234567... street5 11115 1 50
Cloud Service für Tabelle 2: -- Service auswählen --
-- Service auswählen --
Eucalyptus
Amazon
Cloud Anbieter übernehmen
© 2013 Jens Kohler, Hochschule Mannheim
Impressum Kontakt Hilfe


SeDiCo - Cloud Konfigur
localhost:8080/SeDiCoPHP/table_config.php
1 Cust1 fname1 2001-01...
2 Cust2 fname2 2002-02...
3 Cust3 fname3 2003-03...
4 Cust4 fname4 2004-04...
5 Cust5 fname5 2005-05...
OS Linux x86
RAM 7 GB
CPU Kerne 2
CPU Takt pro Kern 1.4 GHz
Server 2: Amazon
amazon web services
1 1234567... 1234567... street1 11111 1 10
2 1234567... 1234567... street2 11112 0 20
3 1234567... 1234567... street3 11113 1 30
4 1234567... 1234567... street4 11114 0 40
5 1234567... 1234567... street5 11115 1 50
OS Linux x64
RAM 7 GB
CPU Kerne 8
CPU Takt pro Kern 1.6 GHz
Konfiguration übernehmen