Bridging SQL and NoSQL

May 11, 2012

Author: John Roijackers
john@roijackers.net

Supervisor and tutor: Dr. G.H.L. Fletcher
g.h.l.fletcher@tue.nl

Assessment committee: Dr. T.G.K. Calders
Dr. G.H.L. Fletcher
Dr. A. Serebrenik
A recent trend towards the use of non-relational NoSQL databases raises the question where to store application data when part of it is perfectly relational. Dividing data over separate SQL and NoSQL databases implies manual work to manage multiple data sources. We bridge this gap between SQL and NoSQL via an abstraction layer over both databases that behaves as a single database. We transform the NoSQL data to a triple format and incorporate these triples in the SQL database as a virtual relation. Via a series of self joins the original NoSQL data can be reconstructed from this triple relation. To avoid the tedious task of writing the self joins manually for each query, we develop an extension to SQL that includes a NoSQL query pattern in the query. This pattern describes conditions on the NoSQL data and lets the rest of the SQL query refer to the corresponding NoSQL data via variable bindings. The user query can automatically be translated to an equivalent pure SQL query that uses multiple copies of the triple relation and automatically adds the corresponding join conditions. We describe a naive translation, where for each key-value pair from the NoSQL query pattern a new triple relation copy is used, and present several optimization strategies that focus on reducing the number of triples that must be shipped to the SQL database. We implement a prototype of the developed theoretical framework using PostgreSQL and MongoDB for an extensive empirical analysis in which we investigate the impact of the translation optimizations and identify bottlenecks. The empirical analysis shows that a practical prototype based on the constructed, non-trivial, and generically applicable theoretical framework is possible and that we have developed a hybrid system to access SQL and NoSQL data via an intermediate triple representation of the data, though there still is enough space for future improvement.
This thesis is the result of my master project, which completes my master Computer Science and Engineering at the Eindhoven University of Technology. The project was done within the Databases and Hypermedia group of the department of Mathematics and Computer Science, and carried out externally at KlapperCompany B.V., the company that introduced me to the problem and offered resources to perform the project.

I would like to take this opportunity to thank a few people for their support during my project. First of all, I would like to thank my supervisor, George Fletcher, for his support and guidance during the project and for offering the possibility to work on my suggested research topic.

Furthermore, I thank all involved employees at KlapperCompany B.V. for offering access to their resources required to carry out my project in the form of a dataset, hardware, a pleasant working environment, and general expertise regarding the project context.

Finally, thanks go out to my family, girlfriend, and friends, for their support and understanding during my project. Particularly in the periods of social isolation to meet an impending deadline.

John Roijackers
# Contents

1 Introduction ................................................................. 1
   1.1 Motivation ..................................................................... 1
      1.1.1 Problem origin .................................................... 1
      1.1.2 Gap between SQL and NoSQL .............................. 2
      1.1.3 Real life situation .............................................. 3
   1.2 Problem statement ..................................................... 3
   1.3 Summary and thesis overview ...................................... 4

2 Problem context ............................................................... 7
   2.1 General direction ....................................................... 7
      2.1.1 Hybrid database .................................................. 7
      2.1.2 Approach ............................................................ 8
      2.1.3 NoSQL representation ......................................... 9
      2.1.4 Data reconstruction ........................................... 11
   2.2 Literature review ....................................................... 12
   2.3 Summary ................................................................. 13

3 Theoretical framework for bridging SQL and NoSQL .......... 15
   3.1 Data incorporation .................................................... 15
      3.1.1 Availability of NoSQL data in SQL ....................... 15
      3.1.2 Transformation of NoSQL data to triples .............. 16
   3.2 Query language ....................................................... 18
      3.2.1 Main idea ........................................................... 18
      3.2.2 Syntax description ............................................. 19
      3.2.3 Collaboration with SQL ...................................... 21
   3.3 Architectural overview .............................................. 21
   3.4 Summary ................................................................. 22

4 Query processing strategies ............................................. 25
   4.1 Base implementation ................................................ 25
      4.1.1 Notation ............................................................ 26
      4.1.2 Translation ........................................................ 27
      4.1.3 Selection pushdown ............................................ 29
In this thesis we bring traditional relational databases and non-relational data storage closer together by reducing the developer’s effort required to combine data from both types of databases. This is motivated by the recent trend to use non-relational data storage, which leads to separate databases for a single application and the consequential extra work involved with managing multiple data sources. In this chapter we further introduce and motivate the problem covered in this report. Firstly, Section 1.1 explains the origin of the topic and motivates why it is both a theoretical and a practical problem. Section 1.2 summarizes the introduced problem and provides a list of tasks to be performed in order to create a working solution. Finally, we provide an overview of the contents of the rest of this report and a brief summary of the achieved results in Section 1.3.

1.1 Motivation

Before diving into the exact problem specification, this section introduces the problem and motivates why the topic is worth investigating. Firstly, we discuss the origin of the problem in Section 1.1.1 and explain which developments have led to the existence of the problem. In Section 1.1.2 we then theoretically describe the problem itself and why this should be solved. Finally, Section 1.1.3 describes a practical situation which indicates that it is not only a theoretical problem, but that there is a commercial need to overcome this problem as well.

1.1.1 Problem origin

Although many different types of database systems exist for decades, the relational database is the most famous database system. These relational database systems are developed, used, and optimized for decades and offer a solid solution for data storage in many different areas. Especially in the area of web applications, where the relational database used to be the standard database system for almost every application.

Only recently a new trend is spotted in web development communities. Besides the traditional relational databases like MySQL, PostgreSQL, and Microsoft SQL Server that are commonly used for websites, developers start considering alternative types of database systems for their data storage. Products like CouchDB, MongoDB, Neo4J, Apache Cassandra, memcached, Redis, JADE, and Apache Hadoop are encountered more often in the context of web development. Instead
of conventional relational databases, these are document stores, graph databases, key-value stores, object databases, and tabular data storage systems.

The mentioned products are so-called NoSQL data storage systems, in the sense that relational databases are referred to as traditional SQL systems and the alternatives are not relational databases. In the remainder of this report we adopt this naming and talk about SQL and NoSQL database systems as defined by Definition 1.1 and Definition 1.2.

**Definition 1.1** An SQL database is a traditional relational database which can be queried using SQL.

**Definition 1.2** A NoSQL database is a database that is not an SQL database. Data is not stored in relations and the main query language to retrieve data is not SQL.

### 1.1.2 Gap between SQL and NoSQL

As a result of these new storage possibilities, developers investigate the available alternatives and NoSQL solutions become more commonplace. Frameworks and programming languages include support for these data storage alternatives and the usage of NoSQL databases increases. Partly because NoSQL solutions are better suited to store some types of data, but also because NoSQL has become a buzzword and developers want to be part of this new hype.

This switch to NoSQL storage does however come with a major disadvantage. Decades of research in the area of SQL databases and the resulting performance optimizations are left behind for a switch to relatively immature replacements. There are enough situations in which NoSQL products are the right tool for the job, but many software functionality can be perfectly modeled in terms of relational entities and is well-suited for traditional SQL data storage.

Problems arise when a single software product requires data storage where a part of the data is ideally stored in a NoSQL database, whereas the rest of the data is perfectly relational and thus well-suited for a traditional SQL database. This raises the question what type of data storage must be chosen for the application. As different parts of the data are well-suited for different types of databases, choosing one type of data storage always implies that a part of the data is stored in a less appropriate way.

The obvious compromise is a solution where the relational part of the data is stored in an SQL database, while the non-relational data is stored in NoSQL. However, this creates a challenge for scenarios where data from both sources is required. Queries that have to combine data from both data sources require extra programming to combine the data and present the correct result.

Another drawback of separating the data storage is that the developers need to access different database systems. Each database has its strengths, weaknesses, and query syntax. Currently, there is no standard for NoSQL query languages. Moreover, because NoSQL is a collective noun for a variety of storage types it is doubtful whether it is even possible to create a single query language. In general, using multiple database systems will increase the complexity and reduce maintainability of the application.

Separating relational and non-relational data in SQL and NoSQL databases respectively certainly has advantages. It is however not straightforward to put this into practice without any substantial drawbacks. Bridging the gap between SQL and NoSQL therefore is an interesting research topic and overcoming the mentioned drawbacks would allow developers to use the right storage solution for both parts of data they have to manage.
1.1.3 Real life situation

We came across a company that is facing exactly the problem described in Section 1.1.2. The technological core of the company's activities concerns managing a large collection with different types of products. These products have different suppliers and often do not have the same set of attributes assigned. Also, the products get revised periodically. In this case the product itself remains the same, but only the newest version of the product can be ordered by customers. These revisions can cause a change in the set of attributes of the product. This implies that the set of attributes of a product is not fixed, which means that using a schemaless NoSQL database for the product data can be more convenient than fitting it in an SQL database.

Besides the product information, the company has to store other data related to their daily activities. This mainly is information regarding customers, orders, suppliers, and invoices. This information can be summarized as CRM data, which is relational data and thus, in contrast to the product data, perfectly suited for an SQL database. Note that this situation matches the general theoretical problem described before. Different parts of the data are well-suited for different types of storage solutions.

As also proposed in Section 1.1.2, the company decided to split the data and use both an SQL and a NoSQL solution to store the CRM data and the product information respectively. As a result of this decision developers now need to retrieve data from different data sources in order to implement certain functionality, for example to generate invoices containing product information. Though this is not an insurmountable problem, it has some disadvantages for the developers. Besides the fact that this requires knowledge of both database systems, the data from both sources has to be manually merged in each situation where combined information is needed.

This raises the question if the problem of combining data from different data sources can be solved in another way, such that developers are not bothered with different data sources and can retrieve information as if they access only a single data storage system. In other words, is it possible to find a solution that bridges the gap between SQL and NoSQL by combining them such that the disadvantages of different data storage systems are eliminated. The company’s desire to combine different database systems in a single application indicates that the previously described problem is not just a theoretical one, but that there is an actual practical need to find a solution for bridging the gap between SQL and NoSQL.

Note that we do not describe the company in much detail and only mention the technological aspects of the problem. Although the aim of this project is to cover the topic generically such that ideas, suggestions, or solutions are universally applicable, the company’s specific situation is sometimes used as a guideline in this report. This implies that some decisions are based on the practical application in the company’s situation. References to ‘the company’ in this report aim to point at the situation described in this section. Also, the company cooperates in the project by offering access to their hardware, data, and expertise.

1.2 Problem statement

Section 1.1 introduced the topic of this report and explains how the problem has arisen, why there is a theoretical need to solve it, and that there are companies currently coping with this problem. In this section we try to formulate the exact problem statement and then list the subproblems discussed in the following chapters.

Splitting application data into different database systems has advantages, but also introduces serious disadvantages which we have to overcome. Most of the disadvantages are a result of the fact that
splitting the data also enforces developers to retrieve data from different sources. We attempt to provide a solution such that the benefits of separate data storage systems remain, while the introduced problems are mostly eliminated.

**Problem statement** Dividing application data over SQL and NoSQL storage generates a gap in terms of uniform data access. Currently, no solutions are available to bridge this gap by masking the underlying data separation and taking over the task of managing multiple data sources.

This problem statement naturally implies the project goal to bridge this gap between SQL and NoSQL. Because bridging the gap is not a trivial task, we outline the context of our problem and focus on a specific set of tasks that together provide a solution to the stated problem. Together these tasks provide a brief overview of the approach we take and a breakdown of how we achieve this goal.

**Task 1 Construct a theoretical framework to bridge SQL and NoSQL**

Determining which approach to take and what our general solution should look like is the first step towards our ultimate goal. The theoretical framework should describe how we model the data, what implications this has for developers, and give an architectural overview of how we want to generically bridge SQL and NoSQL.

**Task 2 Provide a formal definition to specify the developed theoretical framework**

The theoretical framework is specified in more detail and formalized. Like the theoretical framework, the formalization should result in a generic solution applicable to arbitrary SQL and NoSQL databases. Therefore, this formalization should be given in a database independent formal language.

**Task 3 Create a prototype implementation of the theoretical framework**

To verify that our theoretical solution is feasible in practice we want to implement a prototype that demonstrates that the proposal can actually be implemented and that the framework can be efficiently applied in a commercial environment.

**Task 4 Conduct a thorough experiment with the prototype and analyze the results**

The experiment is used to empirically analyze the performance of the prototype. Furthermore, we want to use the experiment results to indicate possible bottlenecks in the prototype and use this information for further improvement of the implementation.

**1.3 Summary and thesis overview**

This report is an exposition of the work we did in an attempt to bring SQL and NoSQL closer together. The recent trend towards using NoSQL databases for application data raises the question where to store which part of the data. Separating the data and storing one part in an SQL database and the other part in a more suitable NoSQL database implies extra work for software developers who then have to manage multiple data sources. Tackling this problem is moreover motivated by a company that encountered this exact problem, which indicates that there is a practical need to bring SQL and NoSQL closer together. We want to bridge this gap between SQL and NoSQL by proposing a solution to this data separation problem.
The tasks defined in Section 1.2 are a guideline for this report. Every task has its own chapter dedicated to it, in which we elaborate on the exact problem and describe our proposed way to solve it. This section gives a brief overview of the contents of the report at chapter level and succinctly summarizes the main topic of each chapter.

Firstly, Chapter 2 defines the problem context. Combining SQL and NoSQL is a relatively new research area. Therefore, we describe the general direction we choose to solve the problem, list some alternative approaches, and explore the available literature related to the problem topic. This chapter elucidates our choice to represent NoSQL data as triples and explains how the NoSQL data can be reconstructed from these triples.

Chapter 3 describes the theoretical framework we propose. This includes details about the implementation at a functional level and gives an overview of the framework architecture. This chapter answers Task 1 by introducing the availability of relation $F$ containing the NoSQL triples in the relational database. Furthermore, we present a query language that allows developers to access both data sources in a single statement.

We formalize this theoretical framework using relational algebra in Chapter 4, thereby addressing Task 2. The solution provided in this chapter offers a generic solution for bridging SQL and NoSQL, regardless of which data storage solutions are used. As part of this formal query translation we include some optimization strategies to optimize the performance of our solution.

Next, Chapter 5 focuses on Task 3 and describes how the formally defined framework is implemented. This also includes the experiment setup, which is required to analyze the performance of the prototype. In this chapter we can no longer talk about SQL and NoSQL in general, but choose specific software products. Our implementation uses PostgreSQL as a relational database. Non-relational data is stored in MongoDB, which is a prototypical example of a NoSQL system. Both software choices are motivated and further implementation details are described as well.

The prototype is used for an experiment to compare the performance between different implementations of the proposed framework. The results are analyzed in Chapter 6, following Task 4. This analysis shows that our prototype is an acceptable solution to bridge the gap between SQL and NoSQL, but that there also is space for further improvements.

Lastly, we give an overview of what we have achieved in Chapter 7. We summarize the conclusions that can be derived from the work we did and offer suggestions for future work not covered in this report. These suggestions can be the next steps to improve the developed prototype towards a commercial strength hybrid SQL-NoSQL database.
In the previous chapter we have indicated the problem introduced by the movement towards NoSQL data storage. Ideally we want to store each part of the application data in a type of database designed and suited for that specific type of data. This allows to fully benefit from the advantages the database offers. However, separating application data over multiple databases implies that the data is split and this introduces use cases in which data from both sources has to be combined. Ideally, we do not want to bother the developer with this task and provide a solution which hides the underlying separated data storage to the developer. The solution for this problem can be searched for in many different directions. We therefore first describe what type of approach we want to take in this report in Section 2.1. This includes how the data is handled and how the hybrid result can be obtained by the developer. After we have determined our main solution direction, we search the available literature related to our intended approach. In Section 2.2 we discuss the relevant literature found and sketch an overview of the work done related to our problem. Finally, Section 2.3 summarizes how we want to tackle the data separation problem and gives an overview of the work currently done by others related to this topic.

2.1 General direction

Before we study the available literature to investigate what work has been done or can be used to combine relational and non-relational data, we first outline the project context. We can search for a solution to combining data from multiple databases in different directions. In Section 2.1.1 we therefore begin with an explanation of which general direction we choose to use and describe how an ideal hybrid database would be modeled. Subsequently, we zoom in on the chosen direction and specify our intended approach in more detail in Section 2.1.2 by motivating our decision to include NoSQL data as a relation in the SQL database. Section 2.1.3 then explains how we model the NoSQL data as triples, such that they can be represented as a relation in SQL. How the NoSQL triples can be used to reconstruct the non-relational data is described in Section 2.1.4.

2.1.1 Hybrid database

Most of the problems that occur when storing a part of your data in an SQL database and the other part in a NoSQL database are based on the fact that the data has to be retrieved from different sources. Multiple data sources means multiple technologies a developer must be able to work with in
order to access all data, multiple query languages, and multiple parts of your code dedicated to data retrieval. Moreover, an additional piece of code responsible for combining the data from both sources again is needed to construct the final result.

A single solution to avoid all of these problems would be to create an abstraction layer on top of the SQL and NoSQL databases. This abstraction would ideally be constructed at a level where it is useful independent of the programming language and database systems used for the implementation at hand. With one query language data could then be retrieved from both data sources through this abstraction layer. The abstraction layer is responsible for retrieving the relevant data from the underlying SQL and NoSQL databases, as well as for combining the fetched data into a single query result. Figure 2.1 visualizes this concept.

![Figure 2.1: Desired database abstraction architecture](image)

The abstraction layer basically performs the same task as an ordinary database system. The query is analyzed first, then the data is fetched from the correct data source, and the separate data fragments are returned as a single solution to the query. The proposed abstraction layer thus could act as if it were a database itself.

In Figure 2.1 read and write queries are supported by the abstraction layer. Important to note here, is that this figure shows the desired full functionality of the abstraction. For the remainder of this report we only consider read queries. That is, for writing to the data sources the developer is still required to connect to the underlying SQL or NoSQL database.

### 2.1.2 Approach

The abstraction layer as shown in Figure 2.1 to create a hybrid database that masks the fact that the data is divided over different sources can be implemented in different ways. These different methods can be roughly categorized in one of the following approaches:

1. Separate software layer
2. Load SQL data in NoSQL, either virtual or materialized
3. Load NoSQL data in SQL, either virtual or materialized

A separated software layer should be created such that it acts as an independent database with probably its own query language to support reading data from both data sources. The abstraction layer then has the responsibility to parse the query, determine in which of the underlying SQL and NoSQL databases the required data is located, retrieve the data, and then automatically combine it to return a single result.

The second approach would imply that we use the NoSQL database as the primary database. The SQL data is virtually loaded into the NoSQL database and can be read using the NoSQL query language. This may require an extension to the NoSQL database to add support for accessing the
SQL data. Depending on the specific SQL and NoSQL solutions, transforming structured SQL data to fit in an unstructured NoSQL database should be relatively easy. Most NoSQL products are designed to store schemaless data, and fitting in structured, relational data therefore means moving the data to a less constrained environment.

Another approach is to incorporate the NoSQL data in the relational database. This incorporation is not necessarily achieved by materializing the NoSQL data in the relational database. It can also be an implementation that makes the NoSQL data virtually available and thus does not actually retrieve the NoSQL data until query execution. Similar to the previous approach this means that the developer opens a connection to the SQL database, and can then also read the NoSQL data. The most obvious challenge with this approach is how to represent the unstructured NoSQL data in an SQL database, and thereafter allow the developer to easily construct queries that read data from the NoSQL data source.

On the other hand, using the SQL database as the primary data storage system and including the NoSQL data has important advantages. Firstly, relational databases and SQL are an established standard regarding data storage. In general, it is safe to say that an arbitrary professional software engineer knows SQL and probably even has experience constructing queries. The familiarity with relational databases and SQL in particular make the relational database a good candidate to serve as the main system for a hybrid database, especially with respect to the ease of adoption.

Furthermore, relational databases are used, researched, and optimized for decades. Not only are relational databases well-known, they also have proven to be a mature solution for data storage. Maturity of SQL databases can be viewed in different perspectives. Besides aspects like performance and stability, many relational databases are also easily extensible. Many SQL databases offer possible ways to integrate external data out of the box.

The movement to use NoSQL databases is a trend visible especially the last few years. Until recently, relational databases offered a suitable solution for data storage for most of the problems occurring in software development. Many data can still be perfectly stored in an SQL database, and the hype NoSQL created causes developers to move relational data to NoSQL storage and thereby sacrificing the advantages a relational database offers in favor of the benefit the NoSQL solution offers for a subset of the data.

In case most data can still be perfectly stored in an SQL database, using that as the primary storage solution will reduce the number of queries that actually need to read the NoSQL data. In that case, the hybrid database is used solely as an SQL database and all advantages of relational databases apply to that particular query. In other words, queries that only read relational data are not affected by the abstraction layer.

Besides the advantages the third approach offers compared to the other methods, our goal is to implement a prototype of the proposed solution. Provided the advantages, especially the extensibility of many SQL databases, we take the third approach and focus on including NoSQL data in SQL. Either as a virtual or as a, perhaps partially, materialized view.

### 2.1.3 NoSQL representation

In Section 2.1.2 we motivated our intention to load NoSQL data in an SQL database. Ideally, we want to provide a solution for creating a hybrid SQL-NoSQL database that is generically applicable, and thus independent of the underlying SQL and NoSQL databases. One of the problems of NoSQL is that there is no standard comparable to SQL for relational databases. All SQL databases can be queried using SQL, whereas many NoSQL databases come with their own query language.
As a result of this lack of standardization there is little to no interoperability among different NoSQL systems. Not only is this difficult for developers, who cannot easily work with different NoSQL solutions without the need to explore the possibilities of another query language and data storage system, but it also causes problems for our project. The absence of an established standard for NoSQL storage makes it difficult to provide a generic abstraction layer.

In this section we describe a triple notation to represent arbitrary data. Furthermore, we give examples of how other data formats can be transformed to the proposed triple representation.

Different NoSQL data types assume a different data structures. Graph databases represent their data as a graph, key-value stores can use a hash map to store their data, whereas document oriented databases contain nested sets of key-value pairs. All data representations however have one thing in common. Each part of a data entity is related to another part of that entity in a specific way.

Inspired by RDF\(^1\), we decide to represent NoSQL data as triples describing how different objects of the data are related to another object. RDF is standardized and a W3C recommendation. Given two data objects \(s\) and \(o\), we can use a triple \((s, p, o)\) to describe that object \(s\) has a relation \(p\) to object \(o\). Data objects can be constant values, but also identifiers to describe an object that has a set of properties described as relations to constant objects. To express that Bob is 37 years old, we can use the triples \(\{(h, \text{name}, \text{Bob}) , (h, \text{age}, 37)\}\). Note that we use \(h\) to combine the name and age information.

Using triples provides a standard way to describe data and results in better interoperability between different NoSQL databases when the data is represented in the same way. The main advantage of triples to represent data is the flexibility. Basically, \((s, p, o)\)-triples can be used to describe any type of data. Because of this flexibility arbitrary NoSQL data and even SQL records can easily be described. Relational tables for example, can be described using triples by creating a triple \((s, p, o)\) for each data row attribute describing that record \(s\) has value \(o\) for attribute \(p\).

The \((s, p, o)\)-triples refer to the RDF specification, where the \(s\), \(p\), and \(o\) respectively refer to the terms subject, predicate, and object. We use the triple notation to convert non-relational data to a format suitable to be represented as an SQL relation. This means that the \(p\) part of the triple is used to describe an attribute name, while the \(o\) value contains the value for this attribute. The \(s\) is used to indicate that multiple triples belong together and that they describe different attributes of the same data entity, like the \(h\) we used in the example to describe that Bob’s age is 37. To clarify the intended use of the different triple positions for our use case, we refer to \((s, p, o)\)-triples as \((id, key, value)\)-triples from now on. Or abbreviated, \((i, k, v)\)-triples.

To exemplify the translation of data to triples, we use the relation given in Table 2.1a. This is a small relation that stores the name and age of people, Alice and Bob in this case. As pointed out before, in order to translate this relation to \((i, k, v)\)-triples we transform each attribute and corresponding value to a triple for every record in the relation. The result is presented in Table 2.1b.

Note that the \(i\) attribute in Table 2.1b is filled with values \(i_1\) and \(i_2\). This can be arbitrary values, as long as it is the same for records that belong to the same record in Table 2.1a. In this case, both data entities \(i_1\) and \(i_2\) have values for each key. In case that for example the \(age\) of Alice is unknown, the corresponding triple \((i_1, \text{age}, 18)\) will not be present.

\[
d = \{ \text{name} : \text{Bob}, \text{age} : 37, \text{courses} : \{ \text{code} : 2IM91, \text{grades} : [8, 6] \} \}
\]

Another important property of many NoSQL databases is that nested data is allowed. Graphs can

\(^{1}\)http://www.w3.org/TR/rdf-primer/
2.1. GENERAL DIRECTION

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>37</td>
</tr>
</tbody>
</table>

(a) Relational representation

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₁</td>
<td>id</td>
<td>1</td>
</tr>
<tr>
<td>i₁</td>
<td>name</td>
<td>Alice</td>
</tr>
<tr>
<td>i₁</td>
<td>age</td>
<td>18</td>
</tr>
<tr>
<td>i₂</td>
<td>id</td>
<td>2</td>
</tr>
<tr>
<td>i₂</td>
<td>name</td>
<td>Bob</td>
</tr>
<tr>
<td>i₂</td>
<td>age</td>
<td>37</td>
</tr>
</tbody>
</table>

(b) Triple representation

Table 2.1: Example data transformation between two representations

also be considered nested data when child nodes are viewed as nested information related to the parent node. Consider the following example of a nested document \( d \), with the corresponding triple representation shown in Table 2.2.

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₁</td>
<td>name</td>
<td>Bob</td>
</tr>
<tr>
<td>i₁</td>
<td>age</td>
<td>37</td>
</tr>
<tr>
<td>i₁</td>
<td>courses</td>
<td>i₂</td>
</tr>
<tr>
<td>i₂</td>
<td>code</td>
<td>2IM91</td>
</tr>
<tr>
<td>i₂</td>
<td>grades</td>
<td>i₃</td>
</tr>
<tr>
<td>i₃</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>i₃</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.2: Triple representation of nested document \( d \)

Although we have only transformed a single nested document, we have multiple values for \( i \), the id attribute. Because document \( d \) was nested we use these different \( i_t \) values to show the relation between the different triples. The \( i_t \) values are only used to connect triples and are not present in the original document \( d \). Because this information is only used to provide information about how the triples are connected they can have arbitrary values as long as the correct structure of \( d \) is described. This implies that this data is not stable and thus should not be used in a query for any other purpose than connecting triples.

2.1.4 Data reconstruction

Data represented as triples provides flexibility, but this comes at a price. As can be observed from the example triple relations in Table 2.1b and Table 2.2, a single NoSQL data entity is represented in multiple records. Moreover, there is no guarantee that the relation is sorted in any way. The records related to the same NoSQL data entity might thus be divided arbitrarily distributed over the triple relation records. We must reconstruct the data, so that related records are combined again.

This is where the \( i_t \) values come in. As pointed out in Section 2.1.3, these values have no meaning except that they determine the structure of the data. Records with the same id value represent keys with values that belong to the same data entity, at the same nesting level. An \( i_t \) value for the value attribute in the triple table means that the attributes belonging to \( i_t \) are nested under the key attribute of the record at hand.

As an example, we look at Table 2.1b again. This is a triple relation where both \( i_1 \) and \( i_2 \) are the id values for multiple records. To combine the id, name, and age information, we need to perform two
consecutive self joins and ensure that the \( \text{id} \) values are equal. Let the relation as shown in Table 2.1b be called \( T \), then renaming and theta-joins do the trick.

### Table 2.3: Stepwise data reconstruction example

<table>
<thead>
<tr>
<th>( T )</th>
<th>( \text{id} )</th>
<th>( \text{name} )</th>
<th>( \text{age} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>1</td>
<td>Alice</td>
<td></td>
</tr>
<tr>
<td>( T_2 )</td>
<td>2</td>
<td>Bob</td>
<td></td>
</tr>
<tr>
<td>( T_3 )</td>
<td>1</td>
<td>Alice</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Bob</td>
<td>37</td>
</tr>
</tbody>
</table>

(a) One relation  
(b) Two relations  
(c) Three relations

The relations \( T_1, T_2, \) and \( T_3 \) as shown in Table 2.3a, Table 2.3b, and Table 2.3c respectively, are constructed using relational algebra expressions. These expressions include the renaming steps and the appropriate theta-joins to reconstruct the data and output the original data as a single relation. Below we give the corresponding relational algebra expressions.

\[
T_1 = \rho_{T_1 (\text{id})} (\pi_{v_i} (\sigma_{k_i=\text{id}} (\rho_{T_i (i,k_i,v_i)} (T))))
\]

\[
T_2 = \rho_{T_2 (\text{id}, \text{name})} (\pi_{v_i,v_n} (\sigma_{k_i=\text{id}\land k_n=\text{name}} (\rho_{T_i (i,k_i,v_i)} (T) \bowtie \rho_{T_n (i,k_n,v_n)} (T))))
\]

\[
T_3 = \rho_{T_3 (\text{id}, \text{name}, \text{age})} (\pi_{v_i,v_n,v_a} (\sigma_{k_i=\text{id}\land k_n=\text{name}\land k_a=\text{age}} (\rho_{T_i (i,k_i,v_i)} (T) \bowtie \rho_{T_n (i,k_n,v_n)} (T) \bowtie \rho_{T_a (i,k_a,v_a)} (T))))
\]

Note that relation \( T_3 \) is equal to the original data as introduced in Table 2.1a. In this specific case two self joins suffice to reconstruct the original data entities from the triples relation \( T \). In general, to reconstruct data with \( n \) attributes \( n - 1 \) self joins are required. The work involved to reconstruct the original data from the triple representation is tedious and we have to automate this process as part of our hybrid database solution.

### 2.2 Literature review

Though the NoSQL movement is relatively new and solutions to combine relational and non-relational data are not extensively researched, related work has been performed in the area of NoSQL and combinations of different data types. In this section we give an overview of the current research state of topics related to our intended goal to bring relational and non-relational data closer together.

Firstly, we focus on the discussion on NoSQL storage in available literature. A thorough survey of available non-relational databases is presented in 2009 [Var09]. This includes an overview of NoSQL storage solutions available at the time of writing and addresses the advantages and disadvantages of non-relational data storage. Furthermore, the question whether a NoSQL database is the correct choice given a dataset is discussed. Others have questioned the NoSQL hype by arguing that the problems NoSQL solution claim to solve are not caused by the relational structure of SQL databases [Sto10]. Also, overviews of both SQL and different types of NoSQL solutions are available which include use cases for different database types [Cat10]. Moreover, this describes the trade off that has to be made when choosing a NoSQL solution to store data in terms of sacrificing certain desirable database properties to improve others. Arguments both in favor and against NoSQL database solutions are present in available literature. The general conclusion however is that NoSQL storage fulfills a need, but its use should be carefully considered and decently motivated as it implies sacrificing important properties offered by traditional relational databases.
In our problem case, considerations have led to the conclusion that data should be divided over different database types. Our proposal to create a hybrid database, which hides the underlying data separation, is not new. In the early 1990’s a concept called mediators has been researched [Wie92]. In this context, a mediator is an extra layer on top of your data storage which takes care of more complicated tasks. This includes calculations over the underlying data, data amount management, keeping track of derivable data, and accessing and merging external data. Also, incorporating NoSQL data as a relation in an SQL database is investigated before. A recent example is the inclusion of XML data as a relation, with a query language modified to query over this XML data [GC07].

Also, the idea of storing data as triples is literally decades old. Again in the early 1990’s it has been introduced and named the first order normal form [LKK91]. The first order normal form is a normalization of the database itself. Instead of real triples, a relation is introduced with four columns representing the relation name, key, attribute name, and attribute value. This way, multiple relations can be merged in a single relation. Without the relation name column to store from which relation the data originates, these are in fact triples describing relational data. Other papers describe the same idea for single relations by providing transformation functions to represent a normal, horizontal relation, to a vertical table containing the same amount of information in triple format [ASX01]. Transforming NoSQL data to triples is a specific application of the general data mapping framework, which aims at describing arbitrary mappings over data [FW09]. In general, these mappings do not necessarily aim at obtaining another representation, but can also be used to derive new information or combine available data.

Our generic NoSQL data description using triples has the disadvantage that the NoSQL data is first transformed to triples, and then later reconstructed again. Furthermore, the triple notation is extremely flexible, but for many data the notation is quite cumbersome and not compact. Also, querying the triple data to select data and combining corresponding triples is inefficient compared to working with relations that already have corresponding attributes combined in a single record. The flexibility of the triple notation however is a major advantage that can justify the associated drawbacks. Moreover, optimizing performance of triple datasets is a topic of research and some promising solutions and indexing techniques have been proposed that can help reduce the performance penalty [FB09].

Effectively querying the NoSQL data, represented as triples, means that triples have to be manually combined. Previous research has led to a query language proposal that includes SPARQL queries in traditional SQL and thereby simplifies querying NoSQL data [CDES05, DS09]. Furthermore, the NoSQL part of the query can be automatically translated to an SQL equivalent by applying a semantics preserving translation [CLF09]. Though this last suggestion is originally aimed at materialized triple data that is actually stored in the relational database, the same ideas can be applied when we keep the non-relational data stored on the NoSQL side and transform it to triples on the fly at query execution.

2.3 Summary

Most of the problems introduced when data for a single application is partially stored in SQL, and partially in NoSQL, are a result of the fact that this implies separate data sources. A solution for this problem, which performs the task of combining data from both databases automatically, can be found in different directions. We choose to create an abstraction layer on top of the SQL and NoSQL databases that presents itself as a single database to the developer, thereby ideally completely hiding the underlying separate SQL and NoSQL databases. To combine the data, we include the NoSQL data in the relational database. The main reason to choose this approach is the maturity of SQL databases. As a result, SQL is a well-known standard that most developers already know. Also,
the SQL database can take care of the more complicated tasks such as ordering, aggregation, and grouping.

The NoSQL data must be included as structured data in a relational database, because SQL databases are designed to work with relations instead of schemaless data. To obtain a generically applicable solution we represent the NoSQL data as RDF-like triples. This allows maximum flexibility and can be used to model any type of data. Furthermore, triples are nicely structured and can thus easily be included in an SQL database as a relation. Via a series of self joins on these triples the original NoSQL data can be reconstructed. Because creating the self joins is a tedious task, we have to find a way to automatically construct this query part with the appropriate conditions.

Our brief literature study showed that our goal to create a hybrid framework to combine SQL and NoSQL data in a single system, at least from the perspective of the developer, has been abstractly described decades ago. Proposals for query languages that allow the developer to query the triple data incorporated as a relation in an SQL database have been made. These suggestions form a good starting point for the design of a theoretical framework for our situation in which we want to access the external data on the fly, using a triple representation to provide a generically applicable solution.
Theoretical framework for bridging SQL and NoSQL

The previous chapter described the abstraction layer we wish to add on top of the separated databases to bridge the gap between SQL and NoSQL by generating a hybrid database. To achieve this, we must transform the NoSQL data to a triple representation and incorporate these triples in the relational database. How we achieve this is explained in Section 3.1.

As mentioned in the previous chapter, reconstructing the NoSQL data in the relational database is a tedious task because multiple self joins are required to achieve this. We want to automate this process, which means that we have to find a way to include the NoSQL data conditions in a query language such that the developer does not have to worry about this reconstruction. In Section 3.2 we therefore specify a query language that can be automatically translated to a pure SQL equivalent that takes care of the self joins and corresponding join conditions.

Section 3.3 then shows the architectural overview of our proposed solution. This describes the steps involved during the lifetime of a single user query. A summary of the entire theoretical framework is given in Section 3.4. The developed theoretical framework matches the criteria to complete Task 1 and serves as a basis for a prototype implementation.

3.1 Data incorporation

We use a triple representation to describe arbitrary NoSQL data. How this is incorporated in an SQL database is described in Section 3.1.1. To keep the solution generically applicable, no specific implementation details are provided. Furthermore, in Section 3.1.2 we explain how NoSQL data can be transformed to triples to allow the data to be accessed in the relational database.

3.1.1 Availability of NoSQL data in SQL

With the triple representation to describe NoSQL data generically as explained in Section 2.1.3, we can now further specify how the data from the NoSQL database is included in the relational database. Without going into implementation details, we assume that in SQL there is a relation \( F(\text{id}, \text{key}, \text{value}) \) available containing all triples to describe the NoSQL data we want to incorporate in the SQL database. Note that this may be a virtual relation. That is, it can be implemented as a non-materialized view on the NoSQL data.
For example, the following query would select all age values available in the NoSQL dataset:

\[ \rho R(\text{age}) (\pi \text{value} (\sigma \text{key} = \text{age} (F))) \]

The relation \( F \) can be used in queries like any other relation in the SQL database. This means we can join \( F \) to ‘normal’ relations to combine SQL and NoSQL data in a single query result. Important to notice however, is that the records in \( F \) are retrieved from an external data source, in this case a NoSQL database. To return the records of \( F \), the SQL database has to retrieve the external data in triple format from the NoSQL database.

This implies that the retrieved data has to be communicated to the relational database. Depending on the exact implementation of \( F \) and the method used to retrieve data from the NoSQL source, this might violate the atomicity of the query and consistency of the query result. Imagine an implementation of \( F \) where data is streamed from the NoSQL source at query time. When a query requires a large amount of data to be streamed to the SQL database, for some reason the NoSQL database might not be available. This can possibly result in an empty relation \( F \). Or, even if the implementation functions correctly, as the first part of the data is being streamed, another user can modify the NoSQL data that still has to be transmitted to SQL. Thereby possibly causing inconsistency in the total set of NoSQL data sent to the relational database.

Furthermore, we assume that \( F \) knows to which NoSQL source it should connect and what data to retrieve. In practice the implementation of \( F \) is provided with parameters to specify how the NoSQL database should be used. For readability however, we ignore these parameters and assume \( F \) returns triples of the NoSQL source we want to access. In the remainder of this report we will thus use \( F \) as the relation of triples that represent the NoSQL data. To include connection parameters, this can simply be replaced with \( F(p_1, p_2, \ldots, p_n) \) in order to provide \( n \) parameters.

### 3.1.2 Transformation of NoSQL data to triples

To ‘fill’ relation \( F \) with triples, the NoSQL data must be transformed to triples. We focus on nested key-value structures, as other NoSQL data formats can relatively easily be transformed to this structure. Nested key-value structures are sets of key-value pairs, denoted as follows:

\[ \{k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n\} \]

Note that any value \( v_i \) can be a nested set of key-value pairs again. Hence, a nested key-value structure. An important restriction is that the keys on the same nesting level, with the same ‘parent’, must be unique. That is, the path of keys to follow through the nested key-value structure to identify a specific key-value pair is unique. We later on assume that this property holds for the development of a theoretical framework. If this constraint is violated, our solution cannot be applied.

For convenience we also allow that \( v_i \) values are written as lists. This notation is treated as pure syntactical sugar to represent a nested set of values with sequentially numbered integer keys. This implies that we hereafter do not discuss the list notation separately, but that it will be covered as a nested set. The following equivalence clarifies how a list is handled as a nested set.

\[ k_i : [v_{i,1}, v_{i,2}, \ldots, v_{i,n}] \equiv k_i : \{0 : v_{i,1}, 1 : v_{i,2}, \ldots, n - 1 : v_{i,n}\} \]
To translate a nested key-value structure $s$ to triples, we have to ensure that the triples corresponding to same structure $s$ are given the same id value. As mentioned before, the exact value of this id is not important, as long as it is the same for triples that belong to the same NoSQL data object. Or in this case, the same nesting level in a key-value structure $s$.

Nested sets of key-value pairs are recursively transformed similar to other key-value pairs, except that these triples get a new unique id value. To connect the triples of the parent key-value structure to the nested one, a triple is added describing the relation using the key of the nested key-value set.

To clarify this we provide a formal transformation. The transformation is done using two functions, $\phi$ and $\psi$. The function subscript is used to ensure that triples are given equal id values $i$. Function $\phi_i$ is used to transform a key-value pair, whereas $\psi_i$ expects a set of key-value pairs.

$$\phi_i(p) = \begin{cases} \{(i,p_k,p_v)\}, & \text{if } p_v \text{ is a constant} \\ \{(i,p_k,j)\} \cup \psi_j(p_v), & \text{if } p_v \text{ is a set} \end{cases}$$

$$\psi_i(S) = \bigcup_{p \in S} \phi_i(p)$$

Note that in the definition of $\phi_i$ we use an undefined variable $j$. This represents a new, unique value that is used as the id value for the nested key-value pairs. Using these two functions we can transform each nested key-value structure $s$ to triples via $\psi_{i_1}(s)$.

A small example that includes a list, and thus nested key-value pairs, is given to put the transformation into practice.

$$s = \{ \text{name : Bob, grades : [8,6]} \}$$

This nested key-value structure is equivalent to:

$$s = \{ \text{name : Bob, grades : \{0 : 8, 1 : 6\}} \}$$

Now we can apply the transformation functions to obtain the triple representation:

$$\psi_{i_1}(s) = \bigcup_{p \in s} \phi_{i_1}(p)$$

$$= \phi_{i_1}(\text{name : Bob}) \cup \phi_{i_1}(\text{grades : \{0 : 8, 1 : 6\}})$$

$$= \{(i_1, \text{name, Bob})\} \cup \phi_{i_1}(\text{grades : \{0 : 8, 1 : 6\}})$$

$$= \{(i_1, \text{name, Bob})\} \cup \{(i_1, \text{grades, i_2})\} \cup \psi_{i_2}(\{0 : 8, 1 : 6\})$$

$$= \{(i_1, \text{name, Bob})\} \cup \{(i_1, \text{grades, i_2})\} \cup \bigcup_{p \in \{0:8,1:6\}} \phi_{i_2}(p)$$

$$= \{(i_1, \text{name, Bob})\} \cup \{(i_1, \text{grades, i_2})\} \cup \{(i_2, 0 : 8)\} \cup \{(i_2, 1 : 6)\}$$

$$= \{(i_1, \text{name, Bob})\} \cup \{(i_1, \text{grades, i_2})\} \cup \{(i_2, 0 : 8), (i_2, 1, 6)\}$$

The definitions of $\phi$ and $\psi$ ensure that the data transformation generates a set of triples representing the NoSQL data with a unique id value per nested set of key-value pairs. Recall that the keys on the
same nesting level, with the same ‘parent’, must be unique. From these properties we can deduce that the combination of \text{id} and \text{key} is unique and thus unambiguously refers to a single value in the NoSQL data.

### 3.2 Query language

We include NoSQL data in an SQL database via a triple relation $F$. Using pure SQL it is possible to request data from both the SQL and the NoSQL data source. In Section 2.1.4 we have shown the relational algebra methods required to reconstruct NoSQL data via a series of self joins on the triple relation. A disadvantage is that the work involved to reconstruct the NoSQL data is tedious. Ideally, we would like to automate the generation of the query part to reconstruct the NoSQL data. However, this requires an extension to the query language. This section describes how SQL can be extended to allow developers to more easily manage NoSQL data in the database queries. Firstly, Section 3.2.1 describes the general concept of variable binding using basic graph patterns from SPARQL. This syntax is modified as described in Section 3.2.2 to better fit our use case. In the same section we discuss the advantages and disadvantages of this modified query language. Finally, in Section 3.2.3 we explain how data originating from the NoSQL database can be used in the SQL part of the user query. Together this specifies the query language we propose to query our hybrid SQL-NoSQL database.

#### 3.2.1 Main idea

To easily describe the self joins required to reconstruct the NoSQL data, a compact description of how the triples should be combined is required. Recall from Section 2.1.3 that representing NoSQL data as triples was inspired by RDF. Likewise, our proposed method to compactly query the NoSQL data in an SQL database is inspired by SPARQL\(^1\), the W3C standardized query language for RDF.

The most important part of a SPARQL query is the basic graph pattern. A basic graph pattern is a set of RDF-like triples to describe data, that can contain variables. If the same variable is used multiple times, this means that the corresponding parts of the data should have the same value as well in order to match the basic graph pattern. The values corresponding to the variables in the SPARQL query are bound to the variable. Other parts of the query can use these variable bindings to apply data selections and projections.

\begin{verbatim}
?i name ?name .
?i age ?age .
\end{verbatim}

\textbf{Listing 3.1: Basic graph pattern as used in SPARQL}

As an example, we use the triples in Table 2.1b again. Now consider the following basic graph pattern in Listing 3.1. The pattern says we want to find variable bindings such that the values for \text{name} and \text{age} have the same \text{i} value. For our example triple dataset this results in the bindings presented in Table 3.1. Note that the result is equal to the original data from Table 2.1a if we project on \text{name} and \text{age}, except for the attribute names which are now the variables.

The variable bindings for the basic graph pattern describe how the triples should be combined. We can use the same method to reconstruct the relevant parts of the original NoSQL data by creating a basic graph pattern that results in variable bindings that correctly combine the triples from relation $F$. We use the concept of variable binding from SPARQL as the basis to easily query the triples that represent the NoSQL data in our framework.

\(^1\)http://www.w3.org/TR/sparql11-query/
3.2. QUERY LANGUAGE

### Syntax description

The SPARQL basic graph pattern syntax to query triples has some disadvantages. To illustrate these, we reuse the nested key-value structure $s$ from Section 3.1.2. More specifically, the triple relation representing the result of $\psi_{i_1}(s)$ as shown in Table 3.2.

#### Table 3.2: Triple relation for $s$

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_1$</td>
<td>name</td>
<td>Bob</td>
</tr>
<tr>
<td>$i_1$</td>
<td>grades</td>
<td>$i_2$</td>
</tr>
<tr>
<td>$i_2$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$i_2$</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

The basic graph pattern in Listing 3.2 binds variables such that the NoSQL data is reconstructed if the bindings are represented like in Table 3.1.

```
1  ?i name ?name .
2  ?i grades ?j .
3  ?j 0 ?f .
4  ?j 1 ?s .
```

Listing 3.2: Basic graph pattern with variables

We first change the notation, so that the basic graph pattern is a real set of triples. Since it then no longer are basic graph patterns, we henceforth call it a NoSQL query pattern. The result is shown in Listing 3.3.

```
1  (?i, name, ?name),
2  (?i, grades, ?j),
3  (?j, 0, ?f),
4  (?j, 1, ?s)
```

Listing 3.3: Modified basic graph pattern notation

Imagine we use this pattern to describe which data we want to select from the triple relation $F$. To be able to use the NoSQL data in the other parts of the SQL query, we allow references to the variables in the pattern. In this case for example, we could include a selection condition $F.s = 6$ in the `WHERE` part of the query.

```
1  (#i, name, ?name),
2  (#i, grades, #j),
3  (#j, 0, ?f),
4  (#j, 1, ?s)
```

Listing 3.4: Modified navigational node notation in a basic graph pattern

Recall that the triple relation’s id attribute values contain arbitrary values, only required to connect the triples. There is no reason to use the id for any other reason than ‘navigation’ between different
triples. Therefore, we adjust the notation for this type of variable. Instead of \( ?v \), we denote them as \( #v \). Furthermore, we disallow these navigational variables to be used outside the NoSQL query pattern. Listing 3.4 displays this adjustment.

Note that each triple in the NoSQL query pattern contains at least one navigational variable. Especially if a data object has many attributes, the NoSQL query pattern may contain just as many triples with the same navigational variable to connect them. To avoid this, we combine different NoSQL query pattern triples into a nested key-value set if they have the same navigational variable as their id value. Also, we can nest these key-value pairs, when appropriate, in order to further reduce the overhead in the NoSQL query pattern. For our example, this means that the triples with key values name and grades can be combined, and that \( #j \) can be nested under the \((#i, \text{grades}, #j)\) triple as illustrated in Listing 3.5.

```plaintext
(name: ?name,
  grades: (0: ?f,
           1: ?s)
)
```

Listing 3.5: NoSQL query pattern

In our example NoSQL query pattern there was only one triple with \( #j \) as its value under which we could nest the triples with id value \( #j \). In general however, it is possible that there are multiple triples with the same variable as value to nest the data, as exemplified in Listing 3.6.

```plaintext
(#i, from, #j),
(#i, to, #j),
(#j, name, ?name)
```

Listing 3.6: Basic graph pattern that cannot be represented as a NoSQL query pattern

In this case the \((#j, \text{name}, ?\text{name})\) triple can be nested under both other triples, which results in the other triple still having the \( #j \) as its value. There is no way to indicate that from and to should have equal values using this syntax. Furthermore, we only allow one pattern to be used as a query. This means that we can only specify conditions for one NoSQL data entity, and it also implies that our previous example where nesting can be done in multiple ways is not supported by a NoSQL query pattern. For the empirical analysis of our prototype we take this into account and ensure this notation supports all our test queries. In general however, this is a significant limitation to the expressiveness of the query language and certainly an important point for future improvement.

Note that the NoSQL query pattern notation is almost similar to the nested key-value structure notation we introduced in Section 3.1.2. Instead of a set notation we use normal parentheses, and we allow keys and values to be replaced by variables. Recall from the same section that we used the nested key-value structure to transform NoSQL data to a triple representation. For our query language we now do the reverse, and use the nested key-value notation to describe which triples we want to query and how they are related.

As also described in the previously referred section, keys in the nested key-value structure to describe the NoSQL on the same nesting level, with the same ‘parent’, must be unique. Likewise, we require that keys in the same nested key-value set of a NoSQL query pattern are unique. Though this limits the expressiveness of the query language, it is a minor restrictions. The NoSQL data cannot contain multiple triples for a combination of id and key, so in practice there is no need to construct such a NoSQL query pattern.
3.2.3 Collaboration with SQL

A NoSQL query pattern is used to describe and select which NoSQL data should be selected. Using the variable bindings, we allow the NoSQL data to be used in the other parts of the query as well. The rest of the query is ordinary SQL, and we have to specify how we include a NoSQL query pattern in an SQL query.

If we include the pattern in an SQL query it should be possible to isolate it, such that we can easily transform the NoSQL query pattern to an SQL equivalent. Furthermore, in the rest of the query we want to refer to the variables from the NoSQL query pattern. A method to cover both these properties at once is to treat the NoSQL part of the query as a separate relation as in Listing 3.7.

```
SELECT
  r.name
FROM
  NoSQL(
    name: ?name,
    grades: (0: ?f,
             1: ?s)
  ) AS r
WHERE
  r.f = 8
```

Listing 3.7: An SQL query with a NoSQL query pattern

By including it we can easily isolate the NoSQL query pattern, as it is surrounded by parentheses and a NoSQL keyword. The alias we assign to the NoSQL query pattern allows references to NoSQL data like any other relation attribute used in an SQL query. In our example via $r.name$ and $r.f$, where $?name$ and $?f$ are used in the NoSQL query pattern.

3.3 Architectural overview

With the query syntax developed in Section 3.2 it is easy to isolate the NoSQL part of a user query. This means that we can easily extract the NoSQL query pattern that describes the NoSQL data. This pattern can be translated to an equivalent SQL query which uses the introduced triple relation $F$, meanwhile making sure that the references in the SQL part of the query are updated accordingly. The result is an SQL query that reads both the SQL and NoSQL data. In Figure 3.1 we visualize the complete query workflow including all steps described in the previous sections.

There are still some problems to be solved before this architecture can be used. Firstly, it includes a translation from the NoSQL query pattern to an equivalent SQL fragment, as identified by the yellow area in the figure. This translation should be performed automatically prior to query execution. The inclusion of a NoSQL query pattern in an SQL query is just a convenient notation introduced to obtain better maintainable and more readable user queries. We should still formally specify the translation to the final SQL query.

Furthermore, the resulting SQL query uses relation $F$, often multiple times. This relation is assumed to be available in the SQL database, but the actual implementation is not yet discussed. Depending on how this is implemented, the transformation is probably unique per NoSQL database. This means that for different NoSQL storage solutions, a new transformation to triples has to be implemented. In the architectural overview this is displayed in the green area.
Figure 3.1: Architectural workflow illustrating the life of a query

Though this requires extra work for each NoSQL database a developer wants to use, this has to be implemented only once. Afterwards the user can reuse the translation. Furthermore, this architecture has the advantage that the database system used to store NoSQL data is completely hidden at the developer’s side of the database. As far as a developer is concerned, relation $F$ exists and contains the NoSQL data in a triple representation.

The resulting SQL query is sent to the relational database and executed like a normal SQL query. The blue area in the query workflow overview identifies this step. Note that this is also the step in the query lifetime where the NoSQL data is reconstructed via a series of joins on the triple relations. The result of the SQL query is the final output for the user query.

3.4 Summary

In this chapter we have described the theoretical solution to combine SQL and NoSQL data in a single system to address Task 1 as listed in Section 1.2 to provide a solution to the problem statement. The general approach from the previous chapter is further specified by describing different aspects of that solution in more detail. We incorporate the NoSQL data in the SQL database via a triple relation $F$. How this relation is implemented is intentionally not covered to keep the solution generic and applicable for arbitrary SQL and NoSQL products. Relation $F$ can be used like any other relation in the SQL database. For clarity we assume that only a single NoSQL database is used and that the implementation of $F$ is aware where to find its non-relational data. By including parameters in the specification however, this can be extended to a more realistic situation where the NoSQL data source information is provided via these parameters.

Furthermore, we have defined functions $\phi$ and $\psi$ that transform a nested key-value structure to a set of triples that describes the same data. This transformation is used to incorporate the NoSQL data in triple format in the SQL database as the previously mentioned relation $F$. Via multiple self joins these triples can be used to reconstruct the NoSQL data in the relational database.

Because writing these self joins on relation $F$ manually for each query is inconvenient and thus undesirable, we introduced an extension to SQL that includes a NoSQL query pattern as part of
the query. Inspired by SPARQL, a NoSQL query pattern describes which data from relation \( F \) we want to use. Using variable bindings, the corresponding NoSQL data can be used in the rest of the SQL query as if it are attributes of a normal relation. The result is a user query containing a NoSQL query pattern in an ordinary SQL query, that is translated to an equivalent pure SQL query in which multiple copies of relation \( F \) are used and the self join conditions are automatically added. Lastly, we provided an architectural overview following the query workflow. This gives a global overview of the steps involved from the moment a developer sends a query until the query result is generated. The following chapter is dedicated to explaining how the translation from a user query with a NoSQL query pattern to a pure SQL equivalent can be performed automatically.
To avoid that the developer should manually reconstruct the NoSQL data from the triple relation $F$, we have introduced a NoSQL query pattern in the previous chapter. This pattern can be included in an SQL query to describe which NoSQL data should be selected. These patterns are convenient for the developer, but the language extension requires us to first translate the user query before we have a valid SQL query that can be executed. In Section 4.1 we describe the minimum translation required to obtain an SQL query that produces the correct result, and also describe how selections can be pushed down such that the translation can be improved and the correct result is obtained within a more acceptable time. As described in Task 2 the translations are formally specified to provide a generically applicable solution. We therefore give the translation in relational algebra notation.

The other sections in this chapter describe further improvements which we apply to decrease the average query execution time and to obtain a feasible prototype. Firstly, Section 4.2 explains how projections can be pushed down the same way as we push selections down. This reduces the amount of triples each copy of $F$ in the resulting SQL query contains. Because each copy of $F$ retrieves the same set of triples, we describe a method to retrieve these triples only once and then reuse them at the SQL side of the implementation in Section 4.3. An overview of all steps in the entire translation is given in Section 4.4. Together these steps automatically translate and optimize the NoSQL query pattern in the user query to the series of self joins that lead to the correct reconstruction of the NoSQL data in the relational database.

### 4.1 Base implementation

This section describes the base translation from the user query with a NoSQL query pattern to a pure SQL equivalent. Before we dive into the translation itself, we first describe some notation conventions in Section 4.1.1. The base translation specified in Section 4.1.2 is the minimal work required to obtain an SQL query that results in the correct output. This translation only focuses on the yellow area in the query workflow overview, and thus does not restrict the set of triples shipped from the NoSQL to the SQL database. Therefore, this is quite a naive translation and far from efficient enough to be feasible in practical situations. Therefore, we provide additional query processing strategies to optimize the translation, like the selection pushdown discussed in Section 4.1.3. As a result of the selection pushdown, the amount of triples to represent the NoSQL data is reduced. Therefore, this selection pushdown strategy aims at optimizing the work in the green area of the query workflow. This selection pushdown can be improved by combining selection conditions as explained in Section 4.1.4.
A final addition to the base translation is the constraint derivation described in Section 4.1.5. Like for the selection pushdown, this optimization can also be improved by applying transitivity following the rules provided in Section 4.1.6.

4.1.1 Notation

The NoSQL pattern in the query is a nested set of key-value pairs. Taking the nested structure of the NoSQL pattern into account, we number the pairs as follows:

\[ \text{NoSQL}(k_1 : v_1, k_2 : v_2, \ldots, k_r : (k_{r,1} : v_{r,1}, \ldots, k_{r,m} : v_{r,m}) \ldots, k_n : v_n) \]

This allows us to talk about key-value pair \( t \), which is the specific pair \( k_t : v_t \). We use this notation to describe the translation of the query containing such NoSQL patterns into pure SQL.

Firstly, we introduce \( F(id, key, value) \) to denote the SQL relation containing the NoSQL data transformed to triple pairs. Disregarding the technical implementation of this relation, the relation \( F \) is available in the relational database and we are allowed to use this like any other relation. Because reconstructing NoSQL data from the triples available in relation \( F \) requires multiple self joins, relation \( F \) will be used several times in our SQL query. It is therefore convenient to introduce additional notation to distinguish between the different instances of relation \( F \) as follows:

\[ F_t = \rho_{F_t(i_t,k_t,v_t)}(F) \]

The relation \( F \) is subscripted with \( t \), just like each of its attributes. This avoids naming conflicts due to overlapping attribute names when applying \( \mathcal{R}_\theta \)-operations to perform self joins on relation \( F \). Furthermore, multiple copies of \( F \) are required to reconstruct data from NoSQL by combining the different key-value pairs, which means each copy \( F_t \) of relation \( F \) is used to represent the single key-value pair \( t \).

Depending on the type of both the key and value of the pair, we can add a selection operator to exclude unnecessary triples from \( F_t \) on the SQL side. That is, we do not change the implementation of \( F \), but filter the superfluous triples in the relational database before applying other relational algebra operations. For a constant key value \( k_t \), the condition \( \text{key} = k_t \) can be added. Similarly, if the value of pair \( t \), \( v_t \), is a constant we can add \( \text{value} = v_t \) as a selection condition on \( F_t \). If \( v_t \) is a nested NoSQL constraint, the same rules apply for each key-value pair \( t, r \) nested under \( k_t \). In other words, for the nested key-value pair \( k_{t,r} : v_{t,r} \) which is represented by relation \( F_{t,r} \), we can add selection clauses if either \( k_{t,r} \) or \( v_{t,r} \) is a constant.

<table>
<thead>
<tr>
<th>( F_t )</th>
<th>Constant ( k_t )</th>
<th>Variable ( k_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant ( v_t )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(\sigma_{\text{key}=k_t,\text{value}=v_t}(F)) )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(\sigma_{\text{value}=v_t}(F)) )</td>
</tr>
<tr>
<td>Variable ( v_t )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(\sigma_{\text{key}=k_t}(F)) )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(F) )</td>
</tr>
<tr>
<td>Nested ( v_t )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(\sigma_{\text{key}=k_t}(F)) )</td>
<td>( \rho_{F_t(i_t,k_t,v_t)}(F) )</td>
</tr>
</tbody>
</table>

Table 4.1: Definition of \( F_t \) for a given key-value pair \( t \)

Table 4.1 summarizes how we define \( F_t \) in relational algebra including the selection conditions based on the key-value pair \( t \). Note that \( t \) can be a nested key-value pair \( a, b \), and the same definition of \( F_t \) can be applied. Furthermore, the table clearly shows that when neither the key nor the value is a constant value \( F_t \) only is a renamed version of \( F \). This is inefficient for the query processing, because in that case \( F_t \) contains all triples in \( F \), and thus potentially the entire NoSQL dataset in triple format.
As an example of how $F_t$ is defined, we look at the following NoSQL query fragment, which is constructed according to the query language specified in Section 3.2.

\[
\text{NoSQL}(id : 18, tags : (a : 2))
\]

In our relational algebra notation this implies three relations $F_{id}$, $F_{tags}$, and for the nested key-value pair $F_{tags,a}$. By applying the rules from Table 4.1, these relations are defined as follows:

\[
\begin{align*}
F_{id} &= \rho_{F_{id}(i_{id}, k_{id}, v_{id})} (\sigma_{key=id\land value=18} (F)) \\
F_{tags} &= \rho_{F_{tags}(i_{tags}, k_{tags}, v_{tags})} (\sigma_{key=tags} (F)) \\
F_{tags,a} &= \rho_{F_{tags,a}(i_{tags,a}, k_{tags,a}, v_{tags,a})} (\sigma_{key=a\land value=2} (F))
\end{align*}
\]

Note that since keys in a NoSQL query pattern must be unique per nested set of key-value pairs, each $F_t$ is uniquely named by unfolding the ancestor key names. If the same key name occurs multiple times in different nested key-value sets of the NoSQL query pattern, this might lead to differently named copies of $F$ with the exact same set of selection conditions applied. Applying the same conditions on $F$ results in the same set of triples for these relations. However, the unique combination of id and key values in the triples enables us to correctly reconstruct the NoSQL data. The correct reconstruction of the NoSQL data is part of the translation we describe next.

### 4.1.2 Translation

We use relational algebra to describe the translation of queries written by the user to pure relational queries. The notation introduced in the previous section allows shorter expressions by abstracting selection conditions and renaming of relation attributes. The translation should be applicable for an arbitrary NoSQL fragment. Except for retrieving the NoSQL data as triples from the non-relational database, there is no need to communicate with the NoSQL database to translate the NoSQL query pattern to an SQL equivalent. When looking at the query workflow as presented in Figure 3.1, this base translation therefore occurs in the yellow area.

#### Unnested patterns

Before we take more advanced NoSQL fragments into account, we focus on an unnested NoSQL fragment with constraints on $n$ keys:

\[
\text{NoSQL}(k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n) \text{ AS } r
\]

Recall that for each key-value pair $t$ we have introduced a relation $F_t$. In this case this means we have $F_1, F_2, \ldots, F_n$ available. To reconstruct the NoSQL data we join these relations such that triples that originate from the same NoSQL object are joined together. This means that the id attributes should have the same value, since the NoSQL fragment is not nested and the $F$ relations coming from the same NoSQL object have been given identical id values. Translated to relational algebra this is equal to:

\[
F_1 \bowtie_{i_1 = i_2} F_2 \bowtie_{i_2 = i_3} \cdots \bowtie_{i_{n-1} = i_n} F_n
\]
Because of the notation we introduced, this query applies the correct selection criteria to each relation \( F_t \), renames the attributes and combines the corresponding triples based on their matching attribute. Suppose that our the example NoSQL fragment is taken from the complete user query \( Q \), we can replace the fragment with the relational algebra translation and convert the rest of the query, which is pure SQL, to the corresponding relational algebra notation.

However, in the rest of the query it is likely that the user referred to variables in the NoSQL fragment to use the NoSQL data for additional selections, projections, or even other SQL operations. This means that we must keep track of the variables used in the NoSQL query pattern and use this information to adjust the SQL part of the query accordingly.

We introduce the function \( \text{insert}(v, a) \) defined as:

\[
\text{insert}(v, a) = \begin{cases} 
V_v = \{a\}, & \text{if } V_v \text{ does not exist} \\
V_v = V_v \cup \{a\}, & \text{otherwise}
\end{cases}
\]

We call this function for each key and value in the NoSQL fragment that is a variable, such that we can keep track of the available variables. As an example, consider the following NoSQL fragment:

\[
\text{NoSQL}(id : \exists i, \text{tags} : (\exists j : \exists i))
\]

When we sequentially process the key-value pairs, we start with \( F_{id} \) where the value is a variable. In this case, we would call \( \text{insert}(i, v_{id}) \) since variable \( i \) is used for the value of \( F_{id} \). Since \( V_i \) does not exist yet it is created with initial value \( \{v_{id}\} \). We do not encounter any variables when we process the second key-value pair which creates the \( F_{tags} \) relation. For the next copy of the triple relation \( F \) however, we see two variables. We process the key first, which via \( \text{insert}(j, k_{tags}, ?j) \) sets \( V_j \) to \( \{k_{tags}, ?j\} \). Then the value of this pair causes a call to \( \text{insert}(i, v_{tags, ?j}) \). Now \( v_{tags, ?j} \) is added to \( V_i \) and we now have two sets \( V_i = \{v_{id}, v_{tags, ?j}\} \) and \( V_j = \{k_{tags}, ?j\} \).

Let \( V \) be the set of variables encountered, in this case \( V = \{i, j\} \), then we can replace the references to these variables in the SQL part of the query by an attribute from a triple relation copy as follows:

\[
\forall v \in V Q[r.v := e_v]
\]

With \( e_v \) an arbitrary element of \( V_v \). In other words, we replace each variable \( v \) with an attribute from one of the \( F_t \) relations that corresponds to an occurrence of \( ?v \) in the NoSQL query pattern.

Now the NoSQL fragment has been translated, all relations are correctly joined and variable references in the SQL part of the query have been replaced by appropriate relation attributes. Finally, we add an additional selection condition to the query which puts a constraint on the variables. If the same variable is used multiple times in the NoSQL fragment of the user query, this implies that these values should be equal. Formally and without minimizing the number of equalities this means:

\[
\bigwedge_{v \in V} \bigwedge_{i,j \in V_v} i = j
\]

In case of our example query the selection \( \sigma_{v_{id} = v_{tags, ?j}} \) must be included in the relational algebra expression.
Nested patterns

Including nested NoSQL constraints in this translation is relatively easy. The entire translation can be applied recursively. We only need to properly join the nested triple relation copies with the other relations. Consider the following generic example structure:

\[
\text{NoSQL}(\ldots, k_r : (v_{r,1} : k_{r,2} : v_{r,2}, \ldots, k_{r,n} : v_{r,n}) \ldots)
\]

Because the triple tables are created such that nested data is connected via a value attribute that is equal to the id value of the nested triples, combining the correct triples is just a matter of inserting the correct join condition for the nested set of \(F\) copies. In this case:

\[
\cdots \bowtie_{i_{r-1}=i_r} F_r \bowtie_{v_{r} = i_{r,1}} (F_{r,1} \bowtie_{i_{r,1}=i_{r,2}} \cdots) \bowtie_{i_r = i_{r+1}} \cdots
\]

The dots indicate that the translation is similar to the standard unnested method. The only difference is the \(\bowtie_{v_{r} = i_{r,1}}\) which correctly connects the nested triples. Note that since the nested relations \(F_{r,t}\) have equal id values, this single join constraint is sufficient.

Example

Based on our previous example we demonstrate a simple translation which combines the nested structure with the use of variables:

\[
\text{NoSQL}(id : ?i, \ tags : \ (a : ?i)) \ \text{AS} \ r
\]

With relations \(F_{\text{id}}, F_{\text{tags}}, \) and \(F_{\text{tags, a}}\) this translates to the following relational algebra query:

\[
\sigma_{v_{\text{id}} = v_{\text{tags, a}}} (F_{\text{id}} \bowtie_{i_{\text{id}} = i_{\text{tags, a}}} F_{\text{tags}} \bowtie_{v_{\text{tags, a}} = i_{\text{tags, a}}} (F_{\text{tags, a}}))
\]

In the SQL part of the query all occurrences of \(r.i\) are replaced with either \(v_{\text{id}}\) or \(v_{\text{tags, a}}\).

4.1.3 Selection pushdown

The translation given in Section 4.1.2 results in correctly reconstructed NoSQL data combined from the different NoSQL triple relations. The resulting relational algebra expression adds a selection condition on each \(F_t\) relation and reconstructs the NoSQL data according to the original structure via a series of joins to combine the correct triples.

In practice however, there is a significant disadvantage of this naive method. The \(F_t\) copies are separate instances of the NoSQL data represented as triples. The NoSQL data is external data if we access it through our relational database. This means that each \(F_t\) relation we use implies that all NoSQL data must be transformed to triples, shipped to the SQL database and at that point we apply the \(\sigma\)-operator to filter the unnecessary triples. This has a significant performance impact, especially when the selection condition is strict and only a small amount of triples is eventually used on the SQL side.

If the implementation of the triple relations would ensure that only triples from NoSQL data matching the selection conditions on the specific \(F_t\) are returned, this should increase the performance of the
entire query. Our running example in this section will be the set of NoSQL data presented in Table 4.2. Note that in this particular case the NoSQL data is perfectly structured, unnested and in fact would have been perfectly suitable to be stored in a relational database. For the purpose of explanation however, there is no need to make the example unnecessarily complicated and this dataset suffices.

<table>
<thead>
<tr>
<th>i</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Example NoSQL dataset

Now consider a query containing the following NoSQL fragment:

\[
\text{NoSQL}(x : 2, y : 1)
\]

In case we would apply the translation described before, this means that we would have relations \(F_x\) and \(F_y\) both defined by a selection operator on the same base relation \(F\). This relation \(F\) is the triple representation of the NoSQL data, and is presented in Table 4.3.

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Example NoSQL dataset transformed to triples

Note that this relation contains 9 records that have to be constructed, communicated to the relational database, and are then filtered by the respective selection operators on \(F_x\) and \(F_y\). In total we have to process 18 records to construct our final result.

We introduce a parameter \(c\) for the relations \(F_t\). This parameter is used to push selection conditions down, such that the implementation that is responsible for retrieving \(F_t\) can take this condition into account. Since \(c\) represents the condition, the value defaults to true if it is not provided. In other words, we can now use \(F_t([c = \text{true}])\) to represent a copy of the triple relation with the appropriate \(\sigma\)-operator applied, as specified before.

This selection pushdown is particularly relevant when the key \(k_t\) of key-value pair \(t\) is a constant. If its corresponding value \(v_t\) is a constant as well, we can push the entire selection \(k_t = v_t\) down. Otherwise, if \(v_t\) is a variable or a nested NoSQL query, we at least know that the key \(k_t\) must exist in the NoSQL data. Therefore, we can still push a selection down, in this case the constraint that \(k_t\) should exist. Formally:
4.1. BASE IMPLEMENTATION

\[
c = \begin{ cases}
  k_t = v_t, & \text{if both } k_t \text{ and } v_t \text{ have a constant value} \\
  \exists(k_t), & \text{if only } k_t \text{ has a constant value} \\
  \text{true}, & \text{otherwise}
\end{ cases}
\]

Note that we only included the \( c \) parameter as an extra parameter. The implementation is responsible for using the information provided via the parameter, but if it ignores this parameter the \( \sigma \)-operators on the \( F_t \) relations ensure that the result is still correct. Whether or not the parameter is used depends on the implementation of the triple relation. In the query workflow overview the implementation of the triple transformation is concentrated in the green area. The selection pushdown strategy focuses on optimizing this part of the architecture.

To show the impact of the selection pushdown, using the example relation from Table 4.2 again, we look at how the relations look as returned by the implementation if it implements the usage of this additional parameter. For \( F_x \) this means that the corresponding condition \( x = 2 \) can be pushed down, as shown in Table 4.4.

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.4: Relation \( F_x (x = 2) \) with selection pushdown

Similarly, Table 4.5 shows the resulting relation we obtain by pushing the condition \( y = 1 \) down to \( F_y \).

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.5: Relation \( F_y (y = 1) \) with selection pushdown

Both relations now consist of 6 instead of 9 records. In total, the number of records we process to execute the entire query is reduced from 18 to 12. If the selections are strict, this could in practice hugely reduce the number of records originating from NoSQL involved in the query execution.

4.1.4 Combined selection pushdown

The selection pushdown can be taken a step further. Both relations \( F_x (x = 2) \) and \( F_y (y = 1) \) described in Section 4.1.3 still contain records that are not in the final result. Recall the example query:

\[
\text{NoSQL}(x : 2, y : 1)
\]

31
From example NoSQL data in Table 4.2, only the object \{i : 2, x : 2, y : 1\} matches all conditions present in the NoSQL query pattern. While we translate the user query to a relational algebra expression, we can collect all these conditions and combine them in a single condition that describes the entire constraint on the NoSQL data we can derive from the NoSQL fragment. If we use parameter \(c\) to push this entire condition down to each copy of \(F_t\), every individual NoSQL relation has information about the entire query and can already filter data that will be lost in the eventual join on the SQL side anyway.

For our example this would mean that for both relations \(F_x\) and \(F_y\) the parameter \(c\) has the value \(x = 2 \land y = 1\). The result of pushing down the combined selections is presented in Table 4.6.

<table>
<thead>
<tr>
<th>id</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.6: Relations \(F_x(x = 2 \land y = 1)\) and \(F_y(x = 2 \land y = 1)\) with combined selection pushdown

Only 3 records are returned by both \(F_t\) relations. This means a total of 6 NoSQL triple records processed for the entire query execution.

Combining the selection conditions makes the translation of the NoSQL query pattern more complex. Instead of directly translating each key-value pair from the NoSQL query pattern to a triple relation copy with selection conditions, we have to collect and combine all selection conditions before we can generate the selection conditions to be pushed down for each copy of \(F\). On the other hand, the translation already requires us to collect information about used variables in the NoSQL query pattern. This information is used for replacing references to this variable in the SQL part of the query and to potentially include additional selection conditions when the same variable is used multiple times. Collecting the combined set of selection conditions and including it as a parameter therefore does not affect the asymptotic running time of the translation.

### 4.1.5 Condition derivation

We have introduced a parameter \(c\) which is used to push down a selection condition to the NoSQL database. Moreover, even combined conditions can be pushed down. There is however yet another possibility to reduce the amount of records shipped to the relational database. We can take the SQL part of the query into account and derive additional constraints that allow additional selection conditions to be pushed down. Consider the following example NoSQL fragment:

\[
\text{NoSQL}(x : ?i) \text{ AS r}
\]

Now let the \texttt{where} part of the SQL query contain the clause \(r.i < 10\). Applying our translation this would be replaced with \(v_x < 10\), as the variable \(i\) is represented by the value attribute of the \(F_x\) triple relation copy. The only condition pushed down to \(F_x\) however is \texttt{exists}(x). This means that NoSQL objects with values for \(x \geq 10\) are returned as well. Obviously, there is no need to create, communicate, and process these triples since they will be filtered out at the relational database side.

By deriving NoSQL conditions based on the \texttt{where} part of the SQL query however, we can also push these conditions down and further reduce the number of triples returned. In this case, the triple relation used in the final query is referred to as \(F_x(x < 10)\).
Note that we now push a $<$-condition down. By keeping the condition in the \texttt{where} part of the SQL query as well, we again do not depend on the implementation to actually apply this constraint on the triples. If for some reason the implementation does not support or use pushed down selections, the selection conditions in the relational algebra expression ensure that the result is still correct. This allows us to push different types of conditions we derive from the SQL part of the query down to the triple relation implementation.

We focus on a standard set of selection condition types, namely $O = \{=, \neq, <, \leq, >, \geq\}$. To derive these conditions from the \texttt{where} part of the SQL query, we first collect the relation names that are actually abstractions for NoSQL data. In case of the small example query mentioned above, this is the relation $r$. Now for each constraint in the \texttt{where} clause of the user query we check if an attribute from $r$ is involved and if the operator is in $O$. If this is the case we add the derived condition to the set of conditions that is already pushed down for each copy of $F_t$.

### 4.1.6 Transitive condition derivation

Also deriving NoSQL conditions from the SQL part of the query can be taken a step further. More specifically, we can use transitivity of different operator combinations in $O$ to derive additional NoSQL constraints which are not explicitly present in the user query. Take the previously used example again:

\[
\text{NoSQL}(x : ?i) \ AS \ r
\]

Say the query also uses the SQL relation $s$ and two of the \texttt{where} clauses are $r.i = s.id \land s.id < 10$. This essentially adds the constraint $r.i < 10$ on the NoSQL data, only we have to use transitivity to derive this constraint. To handle transitivity we apply the rules in Table 4.7 and push the derived conditions down as well.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b$</td>
<td>$a = c$</td>
<td>$a &lt; b$</td>
<td>$a &lt; c$</td>
<td>$a &gt; c$</td>
</tr>
<tr>
<td>$a = c$</td>
<td>$a \neq c$</td>
<td>$a &lt; c$</td>
<td>$a \leq c$</td>
<td>$a &gt; c$</td>
</tr>
<tr>
<td>$a &lt; b$</td>
<td>$a &lt; c$</td>
<td>$a \leq c$</td>
<td>$a \geq c$</td>
<td></td>
</tr>
<tr>
<td>$a \leq b$</td>
<td>$a = c$</td>
<td>$a \neq c$</td>
<td>$a &lt; c$</td>
<td>$a \leq c$</td>
</tr>
<tr>
<td>$a &gt; b$</td>
<td>$a &gt; c$</td>
<td>$a \leq c$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a \geq b$</td>
<td>$a = c$</td>
<td>$a \neq c$</td>
<td>$a &lt; c$</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Table 4.7: Used set of transitivity rules}
\]

### 4.2 Projection pushdown

In this section we take a better look at which triples returned from NoSQL we actually need to construct records for the final result to answer the user query. The NoSQL query pattern is automatically translated to an SQL alternative which accesses NoSQL data as if it were normal relations containing triples to represent the NoSQL objects. The query written by the user can only refer to variables used in the NoSQL fragment. Essentially, this means that the only data that is needed to construct the final result for the entire query are the triples that are associated with the variables in the NoSQL fragment.

For example, if we take a NoSQL fragment $\text{NoSQL}(k_1 : ?i, k_2 : v_2) \ AS \ r$, the SQL part of the query can only refer to $r.i$. Except for applying the selection condition $k_2 = v_2$ on the NoSQL data before
returning anything, there is no need to ship triples containing any other data than $k_1$ values for
NoSQL objects which match the constraint $k_2 = v_2$. Moreover, if no variable from a NoSQL fragment
is referred to in the SQL part of the query, there is no need to access the NoSQL data source at all.

This observation leads to the concept we call projection pushdown. The set of key-value pairs for
which a triple relation $F_t$ is required to be able to access all needed information from the NoSQL
query pattern in the SQL part of the query is called $I_f$. In other words, for each NoSQL fragment $f$
we search for all occurrences of $f.v$ for some variable $v$ in the SQL part of the user query. For each
found $f.v$ we collect all $F_t$ corresponding to key-value pairs from NoSQL fragment $f$ for which either
$k_t = ?v$ or $v_t = ?v$ and call this set $I_f$.

For example, consider the following NoSQL fragment $f$:

\[
\text{NoSQL}(k_1 : ?i, k_2 : (k_{2,1} : v_{2,1}, k_{2,2} : ?j, k_{2,3} : v_{2,3}), k_3 : v_3) \text{ AS } f
\]

If the user query refers to $f.i$ and $f.j$, this means $I_f = \{F_1, F_{2,2}\}$ since the key-value pairs 1 and 2,2 contain variables which are referenced. To determine if for a given key-value pair $t$ we should include $F_t$ in the translation, we recursively determine if $F_t$ is in $I_f$ or, in case $v_t$ is nested, if there exists a nested key-value pair under $t$ which should be in the translation.

For our example this means we would have $F_1$, $F_2$, and $F_{2,2}$ in the translation. Note that $F_2$ is present because $F_{2,2}$ should be in the translation and $F_2$ therefore is required to ensure the structure of the NoSQL data is maintained. Compared to the base translation, we do not use the relations $F_{2,1}$, $F_{2,3}$, and $F_3$.

By reducing the number of relations that return triples we do not only decrease the amount of data
that has to be shipped from NoSQL to the relational database. The final query will also contain less
relations and thus require less joins to reconstruct the NoSQL data in the relational database.

Besides reducing the number of triple relation copies $F_t$, we can use the information about which
variables are used to reduce the amount of triples within a single triple relation copy as well. Consider
the previous example again. There is no need to include the triples representing the key-value pairs
for $k_3 : v_3$, since we do not use this information in the SQL query.

The unnecessary triples are excluded on the SQL side by the selection and join operators. In fact, the
NoSQL data reconstruction filters the triples and thereby the NoSQL key-value pairs. This is
especially the same as a projection on a subset of attributes. Just like for selections, we can push this
projection down to the implementation that retrieves the triple relations and avoid these triples to be
unnecessarily shipped to the relational database. Because all NoSQL objects that are returned are
based on the pushed down selection criteria the result is still correct.

### 4.3 Data retrieval reduction

With the pushdown of selection and projection criteria we have managed to reduce the amount of
triples that has to be shipped from NoSQL to the relational database. Each triple relation copy $F_t$
however requires a new connection to the external NoSQL database to retrieve a set of triples. This
causes overhead for each $F_t$ we use during the execution of a query.

Furthermore, the query planner might generate a plan in which the same relation is used several times or
could possibly access the data to obtain meta information in order to optimize the plan. Each call to a
triple relation requires a connection, communication with the NoSQL database, transforming NoSQL
data to triples and shipping those triples to SQL. In other words, even if we manage to minimize the number of times a triple relation is used in the final SQL query, there might be circumstances that cause more triples to be sent than anticipated. With a performance penalty as a result.

In Section 4.3.1 we therefore first describe a method that combines all triples in a single relation, such that triples have to be shipped to the relational database only once. We achieve this by using a temporary relation. Instead of translating the user query with the included NoSQL query pattern to a single SQL query, we translate it to two queries to be executed sequentially. The first query collects all triples in a single temporary relation $T$, whereas the second query uses $T$ to construct the result of the user query. To decrease the number of triples stored in the temporary relation, we explain how the selection and projection conditions can be used in Section 4.3.2. This query processing strategy aims at optimizing the query execution itself, by reducing the size of the external data that has to be communicated to the relational database. This optimization therefore mainly focuses on the blue area of the query workflow illustrated in Figure 3.1.

4.3.1 General idea

The temporary relation $T$ is in fact similar to the triple relation $F$ we used before. Without any constraints $T$ contains triples that describe all NoSQL data available. Recall that in our translation we use different copies $F_t$ for each key-value pair $t$ which are all instances of the original relation $F$ with selections and renaming applied to it. Therefore, the only thing we have to change in translation is that instead of $F$ the temporary relation $T$ is used. Because $T$ has the exact same set of triples as $F$ the query result will be the same.

Without knowing the details of how $F$ and $T$ are implemented this seems like an unnecessary additional step, first copying all triples from $F$ to $T$ and than executing the exact same query using this temporary relation. However, creating a temporary table first has some useful benefits.

Firstly, all triples are communicated from the NoSQL database to the relational database only once. In the translation we described so far, each $F_t$ relation had to retrieve the triples matching the selection and projection conditions that are pushed down. With the temporary relation the overhead caused by accessing external data is required only once.

Furthermore, because the triples are stored in the temporary relation $T$, the SQL database is forced to materialize the relation. Though this might cost extra time, $T$ can then in the second query be used as if it were an ordinary SQL relation. That is, at the moment the actual user query is executed the SQL database does not have to fetch external data to construct the result. The data is available on the SQL side and the query planner can treat $T$ as any other relation.

Obviously, a disadvantage of materializing the NoSQL data in a temporary relation is the data consistency which cannot be guaranteed. If for some reason there is a significant period of time between creating $T$ and executing the query over $T$, the data might have changed on the NoSQL side in the meantime. However, data consistency already is hard to guarantee when external data is required for a query. Moreover, many NoSQL solutions cannot guarantee data consistency by itself. It is therefore important to remember that inconsistency can be a problem, but in the context of accessing NoSQL data from SQL this will probably not be a serious issue.

Shipping the NoSQL data to the relational database is done during query execution. Therefore, the effect of reducing the amount of data communicated between the NoSQL and the SQL database can be noticed during the query execution. If less communication is required, the relational database can start reconstructing the NoSQL data faster and the average query execution time should decrease as
a result. The temporary relation usage thus focuses on optimizing the blue area of the query workflow, which is responsible for the actual query execution.

### 4.3.2 Triple reduction

Copying all triples from the external data source \( F \) to \( T \) can be a huge task in case we have a lot of NoSQL data. Similar to the usage of \( F \) without the temporary relation, it is likely that many of the triples stored in \( T \) will not be required to construct the result for the user query. If we can avoid these triples from being inserted in \( T \) in the first place, we can decrease the amount of data that has to be shipped from NoSQL to SQL and thereby improve the overall query performance.

To achieve this, we can reuse the ideas about selection and projection pushdowns, but now apply them to the temporary relation creation. In fact, we can use the exact selection and projection conditions from the user query and apply them for the generation of \( T \). The selection condition that was pushed down to NoSQL described the complete selection for NoSQL as described in Section 4.1.4. The projection pushdown described in Section 4.2 can be applied on \( T \) as well.

Because the selection and projection conditions that were applied on each \( F_i \) copy are now already applied on \( T \), we can remove these parameters in the translation of the actual select query.

### 4.4 Summary

A naive translation from the user query to an SQL query is relatively easy. For each key-value pair from the NoSQL query pattern we include a new triple relation copy \( F \). For each such relation we only use the triples that actually belong to the corresponding key and value from the NoSQL query pattern and thus part of the NoSQL data. A disadvantage of this naive approach is that each triple relation copy contains, and thus retrieves, all data from the NoSQL database formatted as triples. We therefore introduce the concept of selection pushdown to reduce the amount of NoSQL data included in each triple relation copy by putting constraints on which triples are actually needed.

Not only selection conditions can be pushed down, also projections can be taken into account. Key-value pairs in a NoSQL query pattern that do not contain a variable are not used in the relational database and creating a separate triple relation copy for it is thus not required in order to reconstruct the correct part of the NoSQL data in the SQL database. Excluding these unnecessary triple relation copies from the SQL query does not only reduce the communication between NoSQL and SQL, it also requires less joins in the relational database to reconstruct the NoSQL data. This means that this single strategy improves the amount of data that has to be communicated and the number of joins required. Both tasks are significant parts of the total query execution time, and therefore minimizing these aspects can improve the overall system performance.

Moreover, instead of retrieving the same set of triples for each triple relation copy we can use a temporary relation \( T \). We fill this temporary relation with all triples required by the SQL query prior to executing the actual SQL query. The SQL query is then modified to use copies of this temporary relation \( T \) instead of a new instance of \( F \). We now only have to retrieve the triples once, materialize them, and use this temporary relation to answer the user query. Especially for a large NoSQL query pattern, which would require many copies of the triple relation, this can significantly reduce the amount of data that has to be shipped from NoSQL to the relational database.

Together these query processing strategies translate the NoSQL query pattern from the user query to an SQL query. Like the proposed theoretical framework and according to the description in Task 2,
these query processing strategies are described independent of the SQL and NoSQL databases used and they are generically applicable. Using the presented translation strategies we can implement a prototype of the proposed hybrid database and conduct an experiment to analyze the performance.
With a complete theoretical framework and specified query processing techniques we now focus on the following task. This is Task 3 and consists of implementing a prototype of the developed hybrid database that can be used to provide a proof of concept to show that the theoretical framework can actually be implemented. Furthermore, we can use this prototype to empirically analyze the framework and investigate the impact of the optimization strategies previously described. This means that we have to choose a specific SQL and NoSQL database. In Section 5.1 we motivate the choice to use PostgreSQL and MongoDB for the prototype implementation.

The company can facilitate the prototype by allowing us to use their hardware, with specifications given in Section 5.2, and providing data which is a good basis for the dataset we construct for the experiment. In Section 5.3 we describe the datasets we use, which also includes an alternative Twitter dataset. For all datasets we construct queries as described in Section 5.4 to be sure that the queries are similar for each dataset and that different types are covered. These different query types can later be used in the analysis of the results to explain certain behavior or indicate bottlenecks that can serve as a basis for further optimizations.

Section 5.5 then mentions the implementation versions we use in the experiment. These implementations are chosen such that we can see the impact of certain query processing strategies by comparing the different implementations. The overall setup of the experiment is elucidated in Section 5.6, where the performance metric is also defined. In Section 5.7 we summarize the complete experimental framework that is used to obtain the results for the empirical analysis discussed in the next chapter.

5.1 Software

In Chapter 3 we designed a software oblivious framework, and also the query processing strategies discussed in Chapter 4 are generically applicable. To construct a prototype of this framework and be able to empirically analyze the performance of the implementation however, we have to pick specific SQL and NoSQL storage solutions. The software we choose determines the possible ways to implement our framework depending on the available features. Therefore, the software choice is important and arguments to justify the chosen SQL and NoSQL databases are discussed in Section 5.1.1 and Section 5.1.2 respectively.
5.1.1 SQL

For several reasons we use PostgreSQL\(^1\) as the SQL database in our experiment. Firstly, PostgreSQL is a well-known and relatively mature relational database. Ideally, we would like to create a commercial-strength solution. It is therefore important that we create a stable and reliable implementation. With some big names on the featured user list, like Greenpeace, IMDB.com, SourceForge, Apple, Fujitsu, and Skype, it is safe to consider PostgreSQL to be an enterprise class relational database.

Moreover, PostgreSQL is open source and freely available. This implies that we are allowed to use it free of charge. But more importantly, because it is an open source project we are able to analyze possible problems in detail by inspecting the source code if necessary. Furthermore, we could modify or add code ourselves if required for our prototype implementation.

Also, PostgreSQL offers great extensibility. Many extensions for the database are available, but writing new extensions is also possible and because of the available documentation relatively easy. The major benefit this gives us, is that it provides us with different ways to implement the triple relation \(F\). If PostgreSQL does not allow a decent way to achieve this, we can always write our own extension to implement this relation.

Lastly, the company also uses PostgreSQL and has a dataset available stored in PostgreSQL. Although SQL databases cooperate quite well, not having to migrate the data to another data storage system first is an advantage. Moreover, the prototype could be used in the company’s context to see it in action in a business environment.

5.1.2 NoSQL

While relational databases are all relatively similar because of the standardization, and a choice for one of them mainly depends on some specific features we want to use, the spectrum of NoSQL databases is much more diverse. There is no such thing as the NoSQL database, let alone a standard that is adopted by all NoSQL databases. In the theoretical description of our proposed solution we have provided a NoSQL database oblivious framework. Which specific NoSQL storage solution we use is not relevant for the developer when writing a user query.

For the empirical analysis however, we must choose a NoSQL database. This can influence the results, as the transformation to a triple representation can possibly be more difficult for some NoSQL data. The NoSQL database we choose to use is MongoDB\(^2\). This is a document-oriented database. An advantage of this choice is that documents in MongoDB are similar to the nested key-value structure described Section 3.1.2. This means that we can use the \(\phi\) and \(\psi\) functions as defined.

Like PostgreSQL, MongoDB also is a well-known product. Despite its relatively immaturity, the list of users that use MongoDB is impressive. Among others, MTV, SourceForge, Disney, The New York Times, bit.ly, and GitHub are users of MongoDB. Important to note here however, is that most company’s use MongoDB for only a part of their applications. At least this indicates that companies see the potential of NoSQL databases and have started to work with MongoDB to see how it performs in practice.

Another similarity between PostgreSQL and MongoDB is that MongoDB also is open source and freely available. This means that for MongoDB we are also allowed to freely use it for our prototype.

---

\(^1\)http://www.postgresql.org/
\(^2\)http://www.mongodb.org/
and in addition are able to inspect the source code if needed to explain certain behavior. Though
the use of the NoSQL database is restricted to data selection and will therefore probably not be the
source of unforeseen problems during the prototype implementation.

And finally, as with the SQL database choice, the company stores its non-relational data in MONGODB.
Using the same NoSQL solution thus has the advantage that we can use their dataset without a
migration to another NoSQL database.

5.2 Hardware

We run the experiment on a single machine to minimize the communication costs involved in shipping
NoSQL data over to the SQL side. Although these costs are relevant in a real life situation, our
prototype implementation focuses on optimizing the actual processing time required to combine the
SQL and NoSQL data. By neglecting communication costs we reduce the environmental influences
on our results, because network delays can no longer affect the communication costs.

Both the SQL and NoSQL database run on a single server with DEBIAN WHEEZY/SID as its operating
system and the following hardware specifications:

- HP PROLIANT DL380 G7
- 2 x Quad core Intel Xeon E5640 processor
- 36 GB of DDR3 memory
- 4 x 450 GB 10krpm SAS hard disk in RAID 5
- 2 x 60 GB SSD in RAID 1

Due to practical constraints a virtual machine on the same server is responsible for the experiment
execution. The virtual machine also has DEBIAN WHEEZY/SID as its operating system, uses two
processor cores and has 2 GB of memory available. This is more than enough to run a script that
connects to the main machine, sends a query to be executed, and then retrieves and stores the result.
Because the virtual machine is on the same server no actual network communication is required,
although the communication between both ‘machines’ goes via TCP and thus the network stack.

5.3 Data

To get a more detailed picture of how our prototype performs, we take the type of data which we
query into account. This means that we have to make sure that we organize our data such that we are
able identify the datasets and compare the results. To achieve this we use two different data sources.
As described in Section 5.3.1, we create different datasets based on real product data. Because these
datasets are different variants of the same data source, Section 5.3.2 describes a different dataset based
on Twitter data.

5.3.1 Products

The first data source is a set of salable products from the company. The real products as provided
contain more information, but for the purpose of our empirical analysis only a subset of the product
properties is used. The products we use all have the structure shown in Listing 5.1.
Important to note here is that the `location` and `booking_deadline` values are not necessarily available. To be more precise, for both properties independently, they exist only for 50% of the products. The `versions` attribute contains a list of nested version information. This means that for a product with $i$ versions, the value for `versions` is a list containing $i$ sets of properties with indexes 0 until $i - 1$.

Furthermore, the values `title_id` and `concern_id` are integers which correspond to PostgreSQL primary keys from the corresponding `Titles` and `Concerns` relations.

Datasets based on this data source are hereafter referred to as $S_{n,s,j}$. The $n$, $s$, and $j$ parameters determine the exact instance of the dataset and are explained next. All combinations of parameter values are used as separate datasets, which means a total of 12 datasets based on the products data source.

**NoSQL data size**

The value of $n$ indicates the number of documents and thus separate products in MongoDB. The possible values and corresponding number of products can be found in Table 5.1.

<table>
<thead>
<tr>
<th>$n$</th>
<th>NoSQL documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$</td>
<td>100 000</td>
</tr>
<tr>
<td>$h$</td>
<td>400 000</td>
</tr>
</tbody>
</table>

*Table 5.1: Possible values and corresponding meaning of the $n$ parameter*

**SQL data size**

The $s$ represents the number of PostgreSQL records contained in the `Titles` and `Concerns` relations. Like the previous parameter, we use Table 5.2 to show the possible values and exact meaning for this parameter. Note that the number of records in the `Titles` and `Concerns` relations is always the same. So in case $s$ has value $h$, this means that the products can be linked to 10 000 separate titles as well as 10 000 different concerns.
5.3. DATA

<table>
<thead>
<tr>
<th>$s$</th>
<th>PostgreSQL records</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$</td>
<td>1000</td>
</tr>
<tr>
<td>$h$</td>
<td>10 000</td>
</tr>
</tbody>
</table>

Table 5.2: Possible values and corresponding meaning of the $s$ parameter

Join probability

Finally, parameter $j$ is used to indicate the join probability between relational and non-relational data. The datasets are constructed such that this probability is applicable in both directions. This means that given a MONGODB document the probability that a referred Titles or Concerns record exists is equal to the probability that given a POSTGRESQL record it is coupled to a product in MONGODB. An overview of these probabilities is given in Table 5.3.

<table>
<thead>
<tr>
<th>$j$</th>
<th>Join probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$</td>
<td>0.05</td>
</tr>
<tr>
<td>$m$</td>
<td>0.20</td>
</tr>
<tr>
<td>$h$</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 5.3: Possible values and corresponding meaning of the $j$ parameter

5.3.2 Tweets

Since all 12 product datasets described in Section 5.3.1 are based on the same source, we want to add a dataset constructed from a different source to the comparison. The Twitter Search API\(^3\) returns data in a JSON format. Furthermore, the returned Twitter messages contain nested information and have properties that do not always exist. It is therefore useful as a data source and we can create a coherent dataset by collecting tweets based on the same search query.

We only use a subset of the Twitter message information that is returned by the Twitter Search API. The structure of the tweet information that we store is shown in Listing 5.2.

```markdown
1 {  
2   created_at: ?c,  
3   iso_language_code: ?l,  
4   text: ?t,  
5   user_id: ?i,  
6   entities: {  
7     urls: [  
8       {  
9         url: ?u  
10       },  
11     ]  
12   },  
13   user_mentions: [  
14     ?m,  
15   ]  
16 }  
17 }
```

Listing 5.2: Tweet data structure

\(^3\)http://dev.twitter.com/docs/api/1/get/search
As visible in this structure, the results returned by the Twitter Search API contain information derived from the text attribute in the nested entities property. In this case, we decided to keep the information about links and mentioned users. It is however possible that a Twitter message does not contain a URL or user mention, in which case the entities does not exist for that tweet in our dataset.

Similar to the product datasets, we again store a part of the data in PostgreSQL. In this case, we create the relations iso_language_codes and Users that are referred to by the values of the iso_language_code, user_id, and user_mention fields in the dataset.

To obtain a coherent dataset we collected 500,000 tweets about a single subject. The Twitter Search API only returns tweets from the last 6 to 9 days, depending on the amount of tweets sent. We must be sure that the search term matches 500,000 tweets in roughly the last week. To ensure that we can indeed find this amount of tweets about a single subject we used the search term ‘bieber’, as Justin Bieber is an almost continuously trending topic and the actual content of the tweets is not relevant for the experiment results. The 500,000 tweets are collected in week 10, and thus between March 5 and March 11, of 2012. We use $S_t$ to denote this Twitter dataset.

5.4 Queries

Not only the different datasets as described in Section 5.3 can influence the performance of our prototype implementation, also the query itself can be important. Depending on the type of query and the complexity the query planner can generate a more efficient plan. To be able to investigate our results in more detail we create a mixed set of queries. We distinguish different flow classes, described in Section 5.4.1, and per flow class we have different query types as discussed in Section 5.4.2. Finally, in Section 5.4.3 we explain how these different flow classes and query types are used to construct the actual queries which are executed.

5.4.1 Flow classes

The main goal of the hybrid database solution is to easily combine data from different underlying data sources. How we use these different data sources in our query to select data can possibly influence the query execution time. For example, selecting only NoSQL data can be more or less efficient than combining that data with relational data. There are different ways to combine SQL and NoSQL data in a single query. Joining relevant NoSQL data to selected SQL records can be more efficient than combining relational information with the selected NoSQL documents, or vice versa. The difference in this case is the data source on which the query mainly bases its selections. We use the term flow class to indicate how the query is constructed. In Table 5.4 we provide an overview of these different flow classes.

<table>
<thead>
<tr>
<th>Flow class</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_i$</td>
<td>NoSQL</td>
</tr>
<tr>
<td>$F_{ii}$</td>
<td>NoSQL→SQL</td>
</tr>
<tr>
<td>$F_{iii}$</td>
<td>SQL→NoSQL</td>
</tr>
<tr>
<td>$F_{iv}$</td>
<td>SQL→NoSQL→SQL</td>
</tr>
</tbody>
</table>

Table 5.4: Overview of the different flow classes

The table shows four flow types, which cover most of the possible flows. The first flow class, $F_i$,
contains queries only selecting NoSQL data. This basically means using our hybrid prototype as a wrapper to select data from the NoSQL database. The query optimizer is limited in its possibilities to optimize the query because the query uses no SQL relations. Queries in $\mathcal{F}_{ii}$ have selection condition aiming at selecting NoSQL documents and joining some data from an SQL relation. Flow class $\mathcal{F}_{iii}$ on the other hand consists of queries that aim at selecting SQL records and combine those with information from NoSQL documents. Finally, queries in $\mathcal{F}_{iv}$ are similar to queries in $\mathcal{F}_{iii}$, except that they join relational information to the NoSQL documents again.

We focus on these flow classes, although other flows are possible. The most obvious one is selecting SQL records only. However, that would be normal SQL queries that are completely processed by the SQL database without utilizing the hybrid aspect of our prototype. We therefore did not include this flow class. Also, queries that select NoSQL data, join SQL data, and then join NoSQL data again is not included because our datasets are not designed to join NoSQL data in different ways to relational data. Note that in general there are no theoretical or practical limitations that prevent queries in these flow classes to be executed using our framework and implemented prototype.

5.4.2 Query types

Besides the previously described flow class of a query, the query execution time can also depend on the properties of a query. More specifically, how the NoSQL data is used in the SQL query. We distinguish two query properties:

(a) Use all mentioned NoSQL keys in the SQL part of the query

(b) Only query over always existing NoSQL keys

Property (a) can influence the query performance because not using a NoSQL key in the SQL part of the query means that it is only required to filter the NoSQL data, but there is no need to ship that part of the data to the relational database. Queries that require less data to be shipped from the NoSQL storage system to the SQL database can therefore perform better if this property is utilized. Recall that the projection pushdown optimization discussed in Section 4.2 is based on taking advantage of this property.

One of the advantages of MongoDB is the schemaless data storage. This means that attributes we want to use in the NoSQL part of the query do not necessarily exist in the NoSQL document. Property (b) is included to analyze if this has any influence on the query performance. On the NoSQL side, including selection conditions over possibly non-existing attributes might influence the performance and thereby affect the total query execution time.

We create different query types based on the possible combinations of these two properties, as summarized in Table 5.5.

<table>
<thead>
<tr>
<th>Query type</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>$(a) \land (b)$</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>$(a) \land \neg(b)$</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>$\neg(a) \land (b)$</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>$\neg(a) \land \neg(b)$</td>
</tr>
</tbody>
</table>

Table 5.5: Overview of the different query types
5.4.3 Query construction

For each dataset we construct a set of query templates used for the empirical analysis of the prototype. A query template is a combination of a flow class and a query type. This means that for each dataset we have a total of 16 query templates. We intentionally call this a query template instead of a query, since we include placeholders in these query templates that can be filled with random values. The query templates for both datasets are included in Appendix A.

Based on each query template we create 10 concrete queries by filling in random values. We do this to vary the data that matches the query conditions and thereby decrease the chance that a query template accidentally is very efficient or inefficient because of the random values that were inserted in case each template is only executed once. Furthermore, to reduce the impact of other processes running on the hardware, we evaluate and measure each concrete query 5 times. For the empirical analysis we drop the highest and lowest value from the 5 repetitions, plus the 2 highest and 2 lowest query template averages. This is done to reduce the influence of resource usage by other processes.

5.5 Triple relation implementation

Until now we assumed that a triple relation $F$ is available in the SQL database which contains triples describing the data in the NoSQL database. Now that we have chosen specific SQL and NoSQL databases for the implementation of a prototype, we have to determine how we implement this relation $F$ with triples from MONGODB in POSTGRESQL, our respective choices for a non-relational and a relational database. Section 5.5.1 first describes how functions, and more specifically table functions can be used in POSTGRESQL. An alternative method, a foreign data wrapper, is presented in Section 5.5.2. Moreover, in Section 5.5.3 we discuss a more convenient way to implement a foreign data wrapper in POSTGRESQL using MULTICORN, which is the method we eventually choose to use to implement the triple relation $F$ in the prototype.

5.5.1 Table functions

POSTGRESQL allows users to define new functions that can be used in SQL queries. These user defined functions can be used to implement functionality that is not supported by the standard set of functions POSTGRESQL offers. Not only do functions avoid writing the same functionality repeatedly, they also allow users to generate results that cannot be obtained using regular SQL without functions. The latter is realized in POSTGRESQL by allowing functions to be written in basically any programming language of your choice, in POSTGRESQL called procedural languages.

By default POSTGRESQL supports functions written in PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. It is however possible to create a language extension for your favorite programming language which allows you to implement functions in that specific language. Examples of non-standard procedural languages developed by the POSTGRESQL community are PL/Java, PL/PHP, PL/Lua, PL/R, and even PL/LOLCODE. Functions can therefore benefit from the features the used procedural language offers, such as the use of existing libraries and collecting data from alternative resources. In other words, we would be able to write a function in which we connect to MONGODB and collect required data.

Listing 5.3 shows an example of a user defined function to increment an integer value written in PL/pgSQL. Although this function implements a basic integer operation, it clearly shows how a stored procedure is written in POSTGRESQL. Note that this function returns a single integer. It is
possible to write functions that returns multiple values in a record. When we write a stored procedure to retrieve data from MONGODB however, we must be able to return more than just one value or a single record.

```sql
CREATE FUNCTION increment(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

**Listing 5.3: PostgreSQL function definition**

Fortunately, PostgreSQL allows us to write functions that return a set of records. This special type of stored procedure is called a table function. Table functions can be used in the `from` part of a query and the resulting set of records is treated as any other relation. Selections, projections, sorting and also joins including the result of a table function are allowed by PostgreSQL. In Listing 5.4 we see an example of a table function definition in PL/Python returning records \{(i, i^p) : i ∈ [1, n]\} for given parameter values \(n\) and \(p\).

```sql
CREATE FUNCTION powers(n integer, p integer) RETURNS SETOF record AS $$
for i in range(1, n + 1):
    yield (i, i ** p)
$$ LANGUAGE plpythonu;
```

**Listing 5.4: PostgreSQL table function definition**

An example of this table function used in an SQL query is shown in Listing 5.5. We retrieve a set of records containing the numbers 1 to 10 with their respective squared values and order these descending.

```sql
SELECT squared
FROM powers(10, 2) AS (normal integer, squared integer)
ORDER BY normal DESC;
```

**Listing 5.5: PostgreSQL table function usage**

Important to note here is the specification of the table function output in the SQL query. In Listing 5.3 the output type was included in the function definition, but in Listing 5.4 we have chosen not to specify the output format for the purpose of demonstrating the possibilities of stored procedures. This allows us to write normal and table functions with a variable output type. However, the SQL query using the function is then required to specify the result type of the function it uses. This means that prior to the actual query execution we are required to tell PostgreSQL what type of data the function will return.

### 5.5.2 Foreign data wrappers

Another method to load external data into PostgreSQL is via a foreign table. From a user perspective a foreign table is exactly the same as any other relation in a database. It has a schema, contains data records and it can be used in select queries without any restrictions. The only difference a normal user notices is the fact that a foreign table is not writable.

Technically, a foreign table is implemented using a foreign data wrapper. Foreign data wrappers are part of the implementation of the SQL/MED\(^4\) extension of the SQL standard. A foreign data

\(^4\)ISO/IEC 9075-9:2003 specification
wrapper is responsible for the retrieval of data for a foreign table. It is implemented using callback functions for planning, explaining and retrieving data rows for a given query. Retrieving data rows is done using an iterator function which returns the next record according to the schema of the used foreign table. Besides these most important functions, callback functions to begin and end a table scan and a special rescan function have to be implemented.

Because the task of a foreign data wrapper is to manage the content of a foreign table, it closely interacts with the PostgreSQL internals. A foreign data wrapper must therefore be implemented in C and linked to the relevant PostgreSQL libraries. When the foreign data wrapper is compiled it can be loaded as an extension in PostgreSQL. As with procedural languages, there are foreign data wrappers available as a PostgreSQL extension written by volunteers in the PostgreSQL community.

Examples are foreign data wrappers for MySQL and Oracle, which allow a user to add a foreign table of which the records are actually stored in another relational database located elsewhere. More applicable to our use case however are foreign data wrappers for other data storage types besides relational databases, like Redis and CouchDB. Even less obvious data sources like LDAP, IMAP and Twitter can be accessed via available foreign data wrappers. The main challenge is to represent the external data source as a relation such that it can be accessed via queries over a foreign table.

A drawback of foreign data wrappers is that they are only recently implemented in PostgreSQL 9.1 and are thus a new feature. Apparently the PostgreSQL developers deem the implementation stable enough for production. However, the immaturity could be a serious disadvantage for companies considering methods to load external data into PostgreSQL. Furthermore, the current foreign data wrapper implementation in PostgreSQL does not contain all functionality specified in SQL/MED. Most importantly, only data selection is supported and foreign tables thus cannot be updated. Also, the current implementation of foreign tables in PostgreSQL is quite primitive. According to the documentation, the query planner does not always optimize queries using foreign tables correctly. Although no details are given in the documentation, we experienced that PostgreSQL does not always take into account that a foreign table requires external data retrieval. Sometimes queries access the same foreign table literally dozens of times, resulting in an enormous communication overhead.

Foreign tables can be passed option arguments at creation, which allow the user to reuse a foreign data wrapper for several foreign tables. Most often these options are used to set connection parameters that allow the foreign data wrapper to connect to the external data source. This can be compared to table function arguments, with the important distinction that those are specified in the query while foreign data wrapper options have to be specified at foreign table creation.

The properties and possibilities of foreign tables seem to be about the same as table functions. Besides pragmatic arguments — unlike table functions, foreign data wrappers are actually intended to be used for accessing external data — there is an important aspect of foreign data wrappers which provide us with a good reason to prefer foreign tables over table functions; the operator pushdown possibilities of foreign data wrappers. It is possible to push SQL operations like selections and projections down to the foreign data wrapper and use this knowledge to exclude unnecessary data records from the foreign table. In other words, foreign data wrappers can take selection and projection conditions into account, use them to select the correct subset of the external data, and thereby reduce the size of the data handled by the SQL query.

In the C code of the foreign data wrapper the selection and projection conditions are provided via the linked PostgreSQL libraries. The foreign data wrapper implementation can use this information to restrict the amount of data that is contained in the foreign table to which the foreign data wrapper belongs. The selection conditions can be used to push down more advanced conditions, like the exists
constraint as described in Section 4.1.3, as well. This is achieved by adding an additional ‘virtual’ attribute to the foreign table. This attribute is virtual because it is not actually populated with NoSQL data. Instead, when a selection condition on this attribute is included in the SQL query, this information is available in the foreign data wrapper. We can thus use virtual attributes to push down more advanced conditions, that cannot be applied on the attributes that contain the actual NoSQL data.

For example, consider the query in Listing 5.6, where \( F(id, key, value, v) \) is a foreign table.

```sql
SELECT id, key, value FROM F WHERE key = 'name' AND v = 'exists(age)'
```

Listing 5.6: Condition pushdown via virtual attribute \( v \)

The virtual attribute is used to provide a constraint on the external data that cannot be expressed otherwise. This is especially useful to implement the combined selection pushdown with more advanced constraints, as it allows us to add constraints on the entire underlying dataset instead of just on the part that is returned. An important detail is that the foreign data wrapper implementation should return a value for the \( v \) attribute that matches the selection condition. In case of our example this means that for each triple we return we add a \( v \) attribute with value \( \text{exists}(age) \). Using a projection on the SQL side this information can be filtered and ignored in the further processing of the query.

Currently only selections and projections can be pushed down to the foreign data wrapper in PostgreSQL, but the SQL/MED specification describes other operators which can be used in the foreign data wrapper. Therefore, in the future it might be possible to push down information regarding sorting, ordering, limiting, or even join conditions to the foreign data wrapper in order to offload a part of the query execution. Also, PostgreSQL currently does not rely on the foreign data wrapper to use the pushed down information and applies the selection conditions on the foreign table records again. Allowing foreign data wrappers to inform the relational database that some operations have already been performed can reduce query execution time in the future.

### 5.5.3 Multicorn

A special foreign data wrapper implementation is MULTICORN\(^5\), which uses the output of a PYTHON script as its external data source. This PYTHON script, in turn, can construct its output using any functionality available in a PYTHON environment. This means that we can write a PYTHON script to access external data and output the resulting records to MULTICORN which forwards the data to POSTGRESQL. Basically, MULTICORN implements the C functions required to be able to use a PYTHON script which manages the actual retrieval of the external data. It is implemented as a normal foreign data wrapper for POSTGRESQL and available to install as an extension. Which PYTHON script MULTICORN should use is specified as an option at creation of the foreign table.

The most important advantage of using MULTICORN is that the actual wrapper can be written in PYTHON instead of C. Writing code to access external data sources in PYTHON is relatively easy compared to writing it in C, in terms of lines of code, readability, and maintainability. Especially for

\(^5\)http://multicorn.org/\n
users that are not experienced C developers, MULTICORN provides a more accessible way to quickly develop your own foreign data wrapper.

However, there is an obvious downside to the advantage of writing the actual data management part of the foreign data wrapper in PYTHON. The foreign table will be less efficient compared to a foreign table using a foreign data wrapper completely written in C. Records returned by foreign tables using MULTICORN have to pass through several abstraction layers. External data is first fetched by a PYTHON script, sent to the MULTICORN C functions, which on their turn ship the data to PostgreSQL, where the records have to be materialized to be able to be used for query processing.

Given our goal to create a proof of concept, we sacrifice raw performance and decide to work with PYTHON and be able to more easily adjust the external data management code. Moreover, the efficiency of the foreign data wrapper implementation is most likely an insignificant detail in our experiment results. Because we are mainly interested in the performance gain from several query processing optimizations, sacrificing some performance in favor of better maintainable code is justified. The foreign data wrapper as used in the experiment, with selection and projection pushdown supported, is included in Appendix B.

5.6 Setup

After describing the software and hardware environment we use for our prototype, elaborating on the data and queries we use for the empirical analysis, and deciding how we implement the triple relation in the SQL database, we now elucidate the experiment setup. Firstly by describing the different implementations we want to compare in Section 5.6.1. The term ‘implementation’ in this case refers to the combination of query processing strategies that are applied to obtain a user query to SQL query translation. In Section 5.6.2 we then explain the implementation details regarding the implementation of the script that runs the experiment and a description of the performance metric we use.

5.6.1 Translation implementation

Recall from Chapter 4 that the implementation of the prototype can be divided into several steps. Starting with a naive implementation we included optimizations to the formal translation. The following list gives an overview of the different query processing steps.

(a) Naive translation as described in Section 4.1.2
(b) Combined selection pushdown as described in Section 4.1.4
(c) Transitive condition derivation as described in Section 4.1.6
(d) Projection pushdown as described in Section 4.2
(e) Temporary table usage as described in Section 4.3

We want to analyze the performance of the hybrid database we propose. Therefore, we include different implementation versions in the experiment so that we can compare them and see the influence of certain implementation steps in terms of query execution times. Table 5.6 provides an overview of the different implementations we compare.

Note that the table contains a column describing the limit of each implementation. After testing the implementation the conclusion was that this limit had to be introduced in order to obtain a result within a reasonable amount of time. A limit of 100 NoSQL data entities is unacceptable for a real
life application. Fortunately, manual testing beforehand indicated that for our final implementation the limit could be increased to 25 000. Though still limited, this is a more feasible size considering that we prototype our theoretical framework.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Steps</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{I}_a$</td>
<td>$(a) \land (b) \land (c)$</td>
<td>100</td>
</tr>
<tr>
<td>$\mathcal{I}_b$</td>
<td>$(a) \land (b) \land (c) \land (d)$</td>
<td>100</td>
</tr>
<tr>
<td>$\mathcal{I}_c$</td>
<td>$(a) \land (b) \land (c) \land (d) \land (e)$</td>
<td>100</td>
</tr>
<tr>
<td>$\mathcal{I}_d$</td>
<td>$(a) \land (b) \land (c) \land (d) \land (e)$</td>
<td>25 000</td>
</tr>
</tbody>
</table>

Table 5.6: Overview of the different prototype implementations

From this table we can see that $\mathcal{I}_a$ implements all steps mentioned in Section 4.1. This is because only implementing the first step required a large amount of data to be communicated to the SQL database, as no constraints are put on the NoSQL data. Step c is also included in the first implementation, because it has a limited effect on the performance of the prototype as this technique is limitedly applicable on our constructed set of queries. This reduces the number of implementations and lets us focus on the major query processing strategy steps to analyze the performance of the prototype.

For the second implementation, $\mathcal{I}_b$, the projection pushdown from Section 4.2 is added to the first implementation. This results in less copies of $F$ used in the translated SQL query and a performance increase should be visible in the results.

Finally, following the specification in Section 4.3, implementation $\mathcal{I}_c$ uses a temporary table to ensure that the triples are only retrieved once and no further communication with the NoSQL database is required. Again, this should reduce the communication and thus improve the prototype performance. Implementation $\mathcal{I}_d$ is identical to $\mathcal{I}_c$, but the limit has been increased to 25 000 NoSQL data entities to see if this implementation can handle more serious amounts of non-relational data.

5.6.2 Details

For the experiment implementation we use POSTGRESQL 9.1.2 as our relational database with the MULTICORN 0.0.9 extension. The foreign data wrapper itself is implemented using PYTHON 2.7.2+. For the NoSQL database we MONGODB 2.0.2 in standalone mode. That is, MONGODB is designed to easily scale over multiple systems. We however use a single server to run both the SQL and NoSQL database.

The experiment itself, responsible for query generation, execution, and managing the result data is written in PHP 5.3.3.7+squeeze8 with Suhosin-Patch. As described in Section 5.2, we use a virtual machine to run this PHP script, while the data processing and hardware intensive database tasks are performed on the actual server. We monitored the resource usage of the script during experiment execution and verified that the virtual machine is not the bottleneck in our empirical setup. While running the experiment memory usage does not exceed 10 MB and processor usage is always below 20 %.

In Figure 5.1 we give an overview of the experimental setup using the theoretical framework summarized in Section 3.3 as a basis. The chosen SQL and NoSQL databases POSTGRESQL and MONGODB are inserted instead of the generic descriptions. The NoSQL part of the user query is a NoSQL query pattern, which is translated to an equivalent SQL fragment using the techniques outlined in Chapter 4. The SQL fragment is merged into the SQL part of the original user query. Note that this step also involves replacing references to variable bindings with the appropriate triple relation copy and attribute name.
The triple relation $F$ is implemented in PostgreSQL using a foreign table. The foreign data wrapper abstraction Multicorn is used to retrieve the NoSQL data from MongoDB and transform it to the correct triple representation according to the function definitions of $\phi$ and $\psi$ as given in Section 3.1.2.

![Figure 5.1: Experimental setup following the query workflow](image)

We measure the performance of the prototype in terms of query execution time. This time is measured in the same PHP script that runs the experiment using calls to the `microtime()` function, available in the standard function library, and computing the difference between several measurements. To be able to analyze the experiment results in more detail we measure multiple aspects of each separate query execution. Firstly, we measure the time required to translate the user query with an included NoSQL query pattern to the SQL equivalent. If applicable, in our setup this means for implementations $I_c$ and $I_d$, we then measure the time required to generate the temporary relation. Finally, we measure the execution time of the actual SQL query including the complete result construction. If not explicitly stated otherwise, we use the total query execution time, which is the sum of the previously mentioned partial execution times, as the performance metric for the empirical analysis.

The execution time breakdown is useful in case we want to determine where the execution time can be seriously reduced or what the bottlenecks of our prototype are. This information is especially useful to indicate possibilities for future optimizations.

### 5.7 Summary

For the empirical analysis we construct an experimental framework to conduct an experiment using a prototype implementation of the previously specified hybrid SQL-NoSQL database solution. For this empirical analysis we have to choose a specific SQL and NoSQL database, which are PostgreSQL and MongoDB respectively. The motivation for these databases is almost similar for both databases. Both PostgreSQL and MongoDB are open source and freely available data storage solutions, which means we can easily use the software to conduct an experiment. Furthermore, using the same databases as the company does saves us a data migration to create datasets based on the company’s product data. The hardware we use is made available by the company.

To cover different types of datasets we use the company’s product data as a basis, and vary the size of
the NoSQL dataset, the size of the SQL dataset, and the join probability. For each combinations of these variables a dataset is generated. Combined with a single Twitter dataset this leads to 13 different datasets. Likewise, we create a set of queries that covers multiple flow classes and several query types within each flow class. In total, we have 16 query templates. Each template contains placeholders for which constant values can be included. Each query template is filled 10 times with random values, and each of these 10 concrete queries is executed 5 times.

The triple relation $F$ is implemented using Multicorn, a foreign data wrapper abstraction which allows us to write the code to retrieve the external data in Python. To run the generated queries we use a PHP script that executes all queries consecutively. The performance metric is the query execution time, which we break down into smaller pieces to be able to analyze it in more detail if desired. We compare 3 different implementations that have other sets of query processing strategies included in the implementation. A fourth implementation is equal to the third implementation, but has an increased limit on the number of NoSQL results. Not only have we addressed Task 3 in this chapter, we also described the experimental environment in which the implemented prototype is analyzed.
Empirical analysis

The experimental framework we created is used to conduct an experiment as explained in the previous chapter and as required to complete Task 4. In this chapter we analyze the results and thereby mainly focus on the impact of optimization strategies we described in Chapter 4. Furthermore, we try to indicate where the bottlenecks are in the prototype implementation so that these can be investigated and optimized in future implementations.

We start our empirical analysis by looking at a global overview of the performance of different implementations in Section 6.1. This gives a global overview of how each implementation performs compared to the other implementations. Then, in Section 6.2 we investigate the base implementation in more detail to see how the results are built up. In Section 6.3 we check if the projection pushdown strategy can indeed increase the prototype performance, and we do the same for the temporary table introduced to minimize communication between NoSQL and SQL in Section 6.4.

Because we had to limit the number of NoSQL results in the base implementation to 100 documents, we compare implementations \(I_c\) and \(I_d\) to see if the NoSQL limit can be increased when all query processing strategies are applied. Recall that \(I_d\) is equal to \(I_c\), except that the NoSQL data limit is set to 25,000 instead of 100. In Section 6.5 we therefore compare the performance of both implementations. This comparison shows the impact of more NoSQL data that has to be shipped to the SQL database, and might therefore indicate possible future optimization opportunities. Section 6.6 briefly summarizes the most important experiment results.

6.1 Result overview

In Section 5.6.1 we outlined the different prototype implementations used in the empirical analysis to compare the impact of different query processing strategies. There are three different implementations, \(I_a\), \(I_b\), and \(I_c\), that we investigate. Respectively, this are the base implementations, adding the projection pushdown to that, and finally also applying the temporary table strategy. The fourth implementation, \(I_d\), has another NoSQL data limit and is only included to investigate the feasibility of the final implementation in a more realistic setting.

For each implementation we have executed a comparable set of queries as described in Section 5.4.3. Recall that we fill each of the 16 query templates 10 times with different values, and run the resulting concrete queries 5 times each. To reduce the influence of external factors, like the server load and availability of resources, we exclude the highest and lowest values for these 5 results. We take the
Likewise, we exclude the two best and worst results per query type. This means that we keep 6 out of the 10 execution results from the randomly filled in query templates. This excludes accidental ‘good’ or ‘bad’ random values used to include in the query templates, resulting in relatively good or bad performance results. The average of these 6 results determines the performance of a single flow class and query type combination for a given dataset and implementation.

In this section we do not go into details, but instead provide a global overview of the empirical results. Figure 6.1 displays the results of the different prototype implementations for each dataset and per query flow class. The average query times indicated in the plot are calculated as the average of the results per query type, where the result per query type is determined as described above. The execution time is the end to end execution time, and thus the sum of the different parts of the query execution time we measure separately.

Figure 6.1: Overview of experiment results per flow class for all implementations with NoSQL data limit 100

Taking into account the logarithmic scale on the vertical axis, the first observation is that implementation $I_c$ performs significantly better than both $I_a$ and $I_b$. For $I_c$ all results are below a second, whereas for the other two implementations not a single result is below a second. The differences between implementation $I_a$ and $I_b$ are smaller, but the effect of the projection pushdown included in
$I_b$ is clearly visible.

Because the NoSQL data is limited to 100 data entities, we can conclude that $I_a$ certainly is not an efficient implementation that could be used in practice. Average query execution times around 20s are unacceptable, especially given the maximum amount of NoSQL data that will be present in the query result. The same conclusion can be drawn for implementation $I_b$, though a few queries take a little over a second and can be considered acceptable. But on average only the performance of $I_c$ appears to be good enough to be used in a practical environment.

In the following sections we study the results in more detail. We analyze what influence the dataset has on the results, what impact the query flow class and query type have, and what the execution time breakdown into the separately measured parts tells us about bottlenecks in the prototype.

### 6.2 Base implementation

The base implementation is the most basic implementation that produces a correct query result within a reasonable amount of time. This includes the naive translation, where each key-value pair in the NoSQL query pattern introduces a new copy of the triple relation $F$ in the SQL query, and the selection pushdown to reduce the number of NoSQL data entities shipped to the relational database.

In Figure 6.2 we see an overview of the prototype performance for implementation $I_a$ per dataset. The plot clearly shows that the time required to translate the user query to a pure SQL query is an insignificant fraction of the total time required to obtain a result, as it is not visible in the bars.

![Average query time for $I_a$ per dataset](image)

**Figure 6.2:** Performance of $I_a$ per dataset

The first thing we notice when looking at the datasets is the average query time for the Twitter dataset $S_t$ of 88.6169 s. Compared to the datasets based on the company’s product data, with an average query time of 14.8419 s, this is a factor 5.97 worse. This is caused by the size of the Twitter data entities, which contain more attributes than the product data, and moreover contain more nested information. As a result, more triples are required to describe the data. Hence, more triples are sent
from the NoSQL to the SQL database which causes a significant decrease in query performance.

For the product datasets we look at the three dimensions separately. Firstly, we see that the number of NoSQL documents, denoted by the first subscript parameter, does not have a real influence on the query performance. This makes sense, as the number of NoSQL documents is limited and thus the total number of NoSQL data entities returned always is at most 100. The only actual difference is the size of NoSQL data on which the selections conditions are applied. Moreover, with 15.0744 s for the lower NoSQL data size versus 14.6093 s for the bigger size the latter is not even slower. This minor difference in performance seems to be caused by the relatively good performance for dataset $\mathcal{S}_{h,l,m}$.

Likewise, the number of records in the SQL relations does not have a significant impact on the prototype performance. Where the average query execution time for a low number of SQL records is 15.3989 s, this is 14.2848 s for a high number of records. A small difference, mainly caused by the relatively high execution time for dataset $\mathcal{S}_{h,l,l}$, which raised the average query time for the datasets with a small SQL data size.

The join probability has a more significant effect on the prototype performance. For a low, medium, and high join probability respectively, the average query execution time for the base implementation $\mathcal{I}_b$ is 18.3599 s, 12.1930 s, and 13.9727 s. The high value for the low join probability is unexpected, as we would expect that a higher join probability leads to more successful joins on the SQL side and thus more data processing compared to the datasets where the amount of data shrinks with each low probability join operation.

The bad performance in case of a low join probability can be an artifact of the product datasets or the query plan generated by PostgreSQL. During the implementation of the experiment we noticed that in some cases PostgreSQL decides to load the same triple relation instance more than once. In fact, sometimes the same triple relation is read dozens of times to answer a single query. Since the data has to be retrieved from NoSQL each time, this might cause the bad performance for low join probability datasets. Moreover, this is an illustration of the disadvantage of the immaturity of foreign data wrappers in PostgreSQL. As stated in the documentation, the query planner is not yet optimized to use foreign tables, and therefore does not take the additional cost of accessing external data into account when creating a query plan.

Now that we have discussed the results for the different datasets, we investigate the influence the query has on the performance. We display the average query execution times for each combination of flow class and query type in Table 6.1. Where the right-most column and the lower row contain flow class and query type averages.

<table>
<thead>
<tr>
<th>$\mathcal{F}$</th>
<th>$\mathcal{Q}_1$</th>
<th>$\mathcal{Q}_2$</th>
<th>$\mathcal{Q}_3$</th>
<th>$\mathcal{Q}_4$</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{F}_i$</td>
<td>4.6360 s</td>
<td>33.9807 s</td>
<td>4.6030 s</td>
<td>23.8295 s</td>
<td>16.7623 s</td>
</tr>
<tr>
<td>$\mathcal{F}_{ii}$</td>
<td>9.8616 s</td>
<td>35.7672 s</td>
<td>9.4285 s</td>
<td>34.7185 s</td>
<td>22.4439 s</td>
</tr>
<tr>
<td>$\mathcal{F}_{iii}$</td>
<td>15.4252 s</td>
<td>48.6681 s</td>
<td>15.2283 s</td>
<td>47.1088 s</td>
<td>31.6076 s</td>
</tr>
<tr>
<td>$\mathcal{F}_{iv}$</td>
<td>11.1810 s</td>
<td>11.2896 s</td>
<td>9.9010 s</td>
<td>12.6431 s</td>
<td>11.2537 s</td>
</tr>
<tr>
<td>Average</td>
<td>10.2759 s</td>
<td>32.4264 s</td>
<td>9.7902 s</td>
<td>29.5750 s</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.1:** Performance of $\mathcal{I}_b$ per flow class and query type

Query types $\mathcal{Q}_1$ and $\mathcal{Q}_3$ perform well compared to $\mathcal{Q}_2$ and $\mathcal{Q}_4$. Recall that $\mathcal{Q}_1$ and $\mathcal{Q}_3$ have in common that the NoSQL query pattern only contains key-value pairs of which the key always exists in the NoSQL data. Apparently prototype implementation $\mathcal{I}_e$ handles user queries with a NoSQL query pattern containing possibly not existing keys very inefficiently. Whether or not we use all NoSQL keys in the SQL part of the query is not relevant for the query performance. This makes sense, because
implementation $I_a$ does not use this information and only naively translates each key-value pair into a separate triple relation copy.

The flow classes $F_1$ and $F_{iv}$ are both quite efficient compared to $F_{ii}$ and $F_{iii}$. Especially flow class $F_{iii}$ is performs badly in combination with the already inefficient query types $Q_2$ and $Q_4$. Recall that flow class $F_{ii}$ aims at selecting NoSQL data and joining SQL data, whereas queries in $F_{iii}$ do the opposite by selecting SQL records and joining NoSQL data. The results show that this is less efficient than $F_1$ and $F_{iv}$. For $F_1$ a possible explanation is that less joins are required on the SQL side, since queries in $F_1$ only use NoSQL data. Queries in $F_{iv}$ on the other hand are more constrained because of the double join with SQL data.

### 6.3 Projection pushdown

We have seen the results for the base implementation $I_a$. For the second implementation, $I_b$, we applied the projection pushdown strategy as explained in Section 4.2. This means that only the NoSQL data corresponding to key-value pairs in the NoSQL query pattern for which the variable binding is used in the SQL part of the query is transformed to triples. This also implies that some triple relation copies that would be in the translation result for $I_a$ are possibly not included in the resulting SQL query for $I_b$ if they are not required to reconstruct the NoSQL data.

In Figure 6.3 we compare $I_a$ and $I_b$ per query type and flow class. We see that for each query type the average query execution time is lower for $I_b$ than for the base implementation $I_a$. Except for $F_{iv}$ and $Q_2$ the projection pushdown improves the average performance of the prototype and can thus be considered an optimization. In some specific cases it even is a huge improvement.

Recall that query types $Q_3$ and $Q_4$ are queries where not all key-value pairs from the NoSQL query pattern are used in the SQL part of the user query. For these query types the performance increase should be higher than for the other query types, because these key-value pairs do not require a corresponding triple relation in the SQL query.

For $Q_1$ the average query execution time is 10.2759s for $I_a$ and 7.1178s for $I_b$, a factor 1.44 improvement. Similarly for $Q_2$, where the $I_a$ average query execution time is 32.4264s versus 21.7875s for $I_b$ and thus a factor 1.49.

The improvement factor for $Q_3$ is higher, 2.48, because the prototype performance increased from 9.7902s to 3.9538s when we applied the projection pushdown strategy. Likewise, the average query execution time for query type $Q_4$ using $I_a$ is 29.5750s, a factor 2.93 slower than the 10.1047s average for $I_b$. These higher factors correspond to the expectation that query types $Q_3$ and $Q_4$ can make better use of the projection pushdown.

### 6.4 Data retrieval reduction

The previous section showed that the projection pushdown is a useful optimization strategy that has most effect when only a small amount of variables in the NoSQL query pattern is used in the SQL query. Another query processing strategy we apply to our prototype implementation is the data retrieval reduction by using a temporary relation $T$ as suggested in Section 4.3. Instead of multiple triple relation copies, all triples are retrieved from the foreign table and stored in a single temporary relation. This relation is materialized on the SQL side and used instead of the separate foreign table copies for the actual SQL query.
This strategy reduces the amount of triples sent from NoSQL to SQL, since they are only shipped once and then reused by the relational database. In particular queries that use many triple relation copies can benefit from this optimization. For our implementation this means that queries with many variables of which the bindings are used in the SQL part of the query can perform better because of the data retrieval reduction.

Another type of query that benefits from this optimization are queries for which a ‘bad’ query plan is created. The PostgreSQL query planner does not fully take into account that foreign tables collect external data and that the query plan should not repeatedly access the foreign table to prevent additional communication costs.

In this section we analyze the effect of the data retrieval reduction strategy by comparing implementation $I_c$ with the previous prototype implementation $I_b$. The query execution times are compared in Table 6.2, where we distinguish between the different flow classes and query types to compare the prototype performance.

The most important observation based on this table is that all average query execution times are below a second for $I_c$. Though we should again remark that the NoSQL data size is still limited to 100 data entities. Compared to the previous implementation $I_b$ however, the prototype performance has been significantly improved. Except for a single case, the average query execution time for $I_c$ is
While more triples are shipped from a limited influence of the join probability is visible in the execution times. Where a low join probability with a medium join probability and 0% join is visible in the plot and it matches the expected behavior. Lower join probabilities imply that not all triples have to be joined if the relational halfway discovers that no other triples match the join.

This appears to be an artifact of the combination of the dataset and the set of constructed queries.

Table 6.2: Performance of $I_c$ per flow class

<table>
<thead>
<tr>
<th>Flow class $F_i$</th>
<th>$I_b$</th>
<th>$I_c$</th>
<th>$I_c / I_b$</th>
<th>$I_b$</th>
<th>$I_c$</th>
<th>$I_c / I_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4.1085</td>
<td>0.4734</td>
<td>0.115</td>
<td>4.8024</td>
<td>0.5187</td>
<td>0.108</td>
</tr>
<tr>
<td>Q2</td>
<td>22.0698</td>
<td>0.6645</td>
<td>0.030</td>
<td>14.7626</td>
<td>0.7037</td>
<td>0.048</td>
</tr>
<tr>
<td>Q3</td>
<td>0.2434</td>
<td>0.3991</td>
<td>1.640</td>
<td>3.1180</td>
<td>0.4453</td>
<td>0.143</td>
</tr>
<tr>
<td>Q4</td>
<td>4.6533</td>
<td>0.4425</td>
<td>0.095</td>
<td>6.7696</td>
<td>0.4254</td>
<td>0.063</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow class $F_{ii}$</th>
<th>$I_b$</th>
<th>$I_c$</th>
<th>$I_c / I_b$</th>
<th>$I_b$</th>
<th>$I_c$</th>
<th>$I_c / I_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10.9477</td>
<td>0.6159</td>
<td>0.056</td>
<td>8.6125</td>
<td>0.6601</td>
<td>0.077</td>
</tr>
<tr>
<td>Q2</td>
<td>38.5469</td>
<td>0.8288</td>
<td>0.021</td>
<td>11.7708</td>
<td>0.8730</td>
<td>0.074</td>
</tr>
<tr>
<td>Q3</td>
<td>5.7486</td>
<td>0.4691</td>
<td>0.082</td>
<td>6.7049</td>
<td>0.4247</td>
<td>0.063</td>
</tr>
<tr>
<td>Q4</td>
<td>18.7359</td>
<td>0.5542</td>
<td>0.030</td>
<td>10.2600</td>
<td>0.5574</td>
<td>0.054</td>
</tr>
</tbody>
</table>

under 15% of the result for $I_b$. Moreover, on average the results are even below 10% of the time required for $I_b$, which means an improvement of more than a factor 10.

An major exception to this observation is the average query execution time for queries of type Q3 in flow class $F_i$, where the use of a temporary is not an improvement. On the contrary, the prototype’s performance has decreased for these queries. Important to note however, is that the average query time of 0.3991 s for implementation $I_c$ still is a good result. In fact, the performance for $I_b$ was significantly above average for this case. Therefore, the data reduction strategy has no effect for these queries and the additional temporary relation generation only required extra time.

Implementation $I_c$ first generates a temporary relation as a separate query, before the query to obtain the actual result is subsequently executed. This allows us to measure the time required for data retrieval separately. In Figure 6.4 we illustrate the division of data retrieval and actual query execution per dataset.

The figure shows that the time required to communicate the data is about the same for the different product datasets. For the more nested Twitter dataset more time is required to ship the data from NoSQL to SQL. The actual query execution for the Twitter data on the other hand is really efficient. This appears to be an artifact of the combination of the dataset and the set of constructed queries. While more triples are shipped from NoSQL to SQL, they can be more effectively joined by the relational database. This leads to a better overall performance for the Twitter dataset compared to the product datasets.

If we analyze the execution time for the product datasets in more detail we see that, like for the base implementation, only the join probability has an impact on the prototype performance. The average query execution time, thus without the data retrieval, is 0.3352 s for a low NoSQL data size versus 0.3296 s for more NoSQL data. No significant difference in performance. Likewise, whether the number of SQL records is low or high also does not have an impact on the time required for execution, given the respective average times of 0.3285 s and 0.3363 s.

A limited influence of the join probability is visible in the execution times. Where a low join probability dataset on average requires 0.3171 s to be processed by the relational database, this takes 0.3214 s with a medium join probability and 0.3587 s with a 100% join. Although the impact is limited, it is visible in the plot and it matches the expected behavior. Lower join probabilities imply that not all triples have to be joined if the relational halfway discovers that no other triples match the join.
condition. Considering that at most 100 NoSQL entities are returned this effect can probably be better observed with a larger set of NoSQL data.

Overall, the data retrieval reduction included in \( \mathcal{I}_c \) is an enormous improvement for our prototype. With average query times under a second this implementation is no longer much too inefficient to be used in practice. In the following section we compare this implementation to \( \mathcal{I}_d \), where we have increased the NoSQL data limit to analyze the prototype performance in a more realistic setting.

### 6.5 Increasing the NoSQL data limit

We have managed to improve our prototype significantly. Where the base implementation \( \mathcal{I}_a \) has an average query time of 20.5225 s, the projection pushdown included in \( \mathcal{I}_b \) brought this down to an average of 10.7474 s. A further improvement is \( \mathcal{I}_c \), with the application of the data retrieval reduction strategy using the temporary triple relation. This brings the average query execution time down to 0.5712 s. However, for all these implementation the number of NoSQL data entities retrieved by the prototype is limited to 100 to keep the average query time, especially for \( \mathcal{I}_a \), within a reasonable range.

To investigate the effect of increasing this limit and analyze whether the prototype can also be used in more realistic settings, our final implementation \( \mathcal{I}_d \) has an increased NoSQL limit of 25 000.

In Table 6.3 we compare the average query times of \( \mathcal{I}_c \) and \( \mathcal{I}_d \) per flow class and dataset. The average query time for all queries in \( \mathcal{I}_d \) is 16.8404 s. Though this is a huge performance decrease compared to \( \mathcal{I}_c \), with 250 times as many results the implementation still performs better than the base implementation \( \mathcal{I}_a \). Furthermore, the performance of the prototype is significantly influenced by the dataset and to a lesser extent the query flow class.

For the product datasets the performance is within reasonable margins for our prototype and experiment setup, but definitely not efficient enough for applications that require commercial strength solutions. Flow class \( \mathcal{F}_1 \) and \( \mathcal{F}_{II} \) show no remarkable results. Both flow classes significantly perform worse with a
higher NoSQL data size. From this observation we deduce that on average less than 25 000 NoSQL data entities match the selection conditions and thus less triples have to be shipped to the relational database.

Flow class $F_{iii}$ and $F_{iv}$ however show some really varying results. We still see that the average query time increases when the join probability increases, but there are some notable outliers. Most striking are the $I_d$ results for datasets $S_{h,l,h}$ and $S_{h,h,h}$ for queries from flow class $F_{iii}$. The relational database seems to have trouble efficiently joining the retrieved triples, since both datasets have a 100% join probability.

The Twitter dataset still performs quite well, with query times below 1.4 s for $F_1$, $F_{ii}$, and $F_{iii}$. For the other flow class, $F_{iv}$, on the other hand, the Twitter datasets performs really badly compared to the product dataset. The good performance for the first three flow classes is caused by the combination of dataset and queries. Because for a majority of the queries much less than 25 000 NoSQL data entities, tweets in this case, matched the selection conditions. The amount of triples that has to be shipped to the relational database is therefore relatively low, which also reduces the amount of processing required to join the triples on the SQL side. The bad result for $F_{iv}$ is most likely caused by a bad query plan and related to the artifacts also visible in the $F_{iii}$ and $F_{iv}$ queries for the product datasets.

The impact of the increased NoSQL data limit is huge. On average the prototype performance is insufficient for industrial strength application. Especially the enormous outliers for some combinations of data and query are undesirable in practical applications. However, this seems to be the result of a bad query plan and can possibly be resolved if we have more control over the query planner and the decisions it makes. Though we have managed to hugely increase the performance of the prototype, for more realistic settings we should look for further optimizations. The most obvious ways to improve the prototype performance are reducing the amount of data communicated between NoSQL and SQL, and techniques to more efficiently join the triples at the relational database side.

Table 6.3: Impact of increasing the NoSQL limit from 100 to 25 000

<table>
<thead>
<tr>
<th>$S$</th>
<th>$F_1$</th>
<th>$F_{ii}$</th>
<th>$F_{iii}$</th>
<th>$F_{iv}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{l,l,l}$</td>
<td>0.5052</td>
<td>9.5917</td>
<td>0.5207</td>
<td>9.4131</td>
</tr>
<tr>
<td>$S_{l,l,m}$</td>
<td>0.4989</td>
<td>8.7164</td>
<td>0.5134</td>
<td>9.2188</td>
</tr>
<tr>
<td>$S_{l,l,h}$</td>
<td>0.5043</td>
<td>8.9252</td>
<td>0.5691</td>
<td>10.1750</td>
</tr>
<tr>
<td>$S_{l,h,l}$</td>
<td>0.4941</td>
<td>9.3733</td>
<td>0.5193</td>
<td>9.7572</td>
</tr>
<tr>
<td>$S_{l,h,m}$</td>
<td>0.4966</td>
<td>9.0614</td>
<td>0.5171</td>
<td>10.2908</td>
</tr>
<tr>
<td>$S_{l,h,h}$</td>
<td>0.5002</td>
<td>7.9555</td>
<td>0.5720</td>
<td>9.9816</td>
</tr>
<tr>
<td>$S_{h,l,l}$</td>
<td>0.4958</td>
<td>13.5595</td>
<td>0.5169</td>
<td>14.3972</td>
</tr>
<tr>
<td>$S_{h,l,m}$</td>
<td>0.4988</td>
<td>13.4030</td>
<td>0.5066</td>
<td>14.7638</td>
</tr>
<tr>
<td>$S_{h,l,h}$</td>
<td>0.5038</td>
<td>12.7724</td>
<td>0.5476</td>
<td>15.0634</td>
</tr>
<tr>
<td>$S_{h,h,l}$</td>
<td>0.5193</td>
<td>13.3464</td>
<td>0.5079</td>
<td>14.7101</td>
</tr>
<tr>
<td>$S_{h,h,m}$</td>
<td>0.4985</td>
<td>13.6935</td>
<td>0.5415</td>
<td>14.7813</td>
</tr>
<tr>
<td>$S_{h,h,h}$</td>
<td>0.4814</td>
<td>13.5785</td>
<td>0.5555</td>
<td>15.1048</td>
</tr>
<tr>
<td>$S_l$</td>
<td>0.4870</td>
<td>1.2257</td>
<td>0.4813</td>
<td>1.3717</td>
</tr>
</tbody>
</table>
6.6 Summary

Using the experimental framework outlined in the previous chapter, we have conducted an experiment to empirically analyze the performance of our prototype and investigate the effects of the different query processing strategies described in Chapter 4. The main goal of the empirical analysis was to compare the different prototype implementations and indicate possible bottlenecks which identify possible future optimization possibilities, as stated in the description of Task 4.

From an average query execution time of $20.5225\text{s}$ for our base implementation $I_a$, we managed to reduce this to $10.7474\text{s}$ by applying the projection strategy, and even significantly improve the performance of implementation $I_c$ by reducing the communication costs using a temporarily materialized triple relation to an average query time of $0.5712\text{s}$.

The query execution time breakdown for the base implementation $I_a$ already indicated that the translation time, to translate the user query with a NoSQL query pattern to a pure SQL query, is negligible. The prototype is really inefficient when querying over the Twitter dataset. For the product datasets queries using possibly non-existing NoSQL data attributes are less efficient. Similarly, queries in flow class $F_{\text{iii}}$, where we mainly select SQL data and join NoSQL information, are less suitable for our base prototype implementation.

With the additional query processing strategy implemented in $I_b$ we were able to improve the performance of the prototype. Especially queries for which many triple relation copies are not required for the reconstruction of the NoSQL data can benefit from the projection pushdown. When we also implement the data retrieval reduction strategy using a temporary relation, the average query execution time is significantly reduced and on average all queries are answered within a second for prototype implementation $I_c$.

When we increase the limit on the number of NoSQL data entities retrieved by the prototype however, the performance of implementation $I_d$ decreases. With average query times of almost $17\text{s}$ however, further optimizations are required to obtain an acceptable performance in a more realistic setting regarding the amount of NoSQL data the prototype can handle. The results indicate that more efficient joins and additional control over the SQL query plan could improve the performance of the prototype.
Conclusions

Our empirical analysis showed that implementing a practical prototype based on the constructed, non-trivial, and generically applicable theoretical framework is possible and that we have developed a hybrid system to access SQL and NoSQL data via an intermediate triple representation of the data. The analysis also showed that there is enough space for future improvement. This chapter concludes this report by summarizing what we have achieved and to what extent we managed to perform the four tasks specified in Section 1.2. The first two tasks were aimed at constructing a theoretical solution to bridge the gap between SQL and NoSQL. How we achieved this is summarized in Section 7.1.

The practical tasks, the last two, are discussed in Section 7.2. We implemented a prototype of the proposed theoretical solution and empirically analyzed its performance. The main results and pointers for future improvement are summed up, followed by a more detailed discussion on suggestions for future work in Section 7.3.

7.1 Summary of theoretical results

To bridge the gap between SQL and NoSQL, we created a hybrid database that acts as a single, read-only database that can be used to retrieve combined relational and non-relational data. Developers using the abstraction layer do not have to worry about the underlying data separation, except which data is stored where. The NoSQL data is transformed to a triple representation. This allows us to model arbitrary data, as we have shown by providing a transformation from arbitrary nested key-value structures to triples.

These triples are incorporated in the relational database via a virtual triple relation $F$. Disregarding the implementation details, this relation retrieves the NoSQL data in triple format when it is accessed by the developer. The triples returned by the introduced triple relation $F$ can be considered key-value pairs, where an identifier in the triple is used to connect the key-value pairs which belong to the same NoSQL data entity. Nested data can be modeled via a key-value pair for which the value is equal to the identifiers of all underlying nested attributes.

On the SQL side we then need to reconstruct the NoSQL data. To achieve this, we can use the identifiers in the triples. Via a series of self joins with the correct join conditions the developer is able to reconstruct the nested NoSQL data in an ordinary SQL query. However, this reconstruction is a tedious task and we wanted to create a convenient method to combine SQL and NoSQL data. We therefore introduce an extension to SQL that includes a NoSQL query pattern in a normal SQL query.
This NoSQL query pattern describes the conditions on the NoSQL data and can furthermore be directly translated to an equivalent pure SQL query with all required joins included. Bound variables according to the NoSQL query pattern can be used in other parts of the SQL to combine the SQL and NoSQL data.

The result of this work is a theoretical framework to bridge SQL and NoSQL. The developer sends a query containing a NoSQL query pattern. This pattern is translated to an equivalent SQL fragment that correctly joins the NoSQL triples retrieved from copies of the triple relation $F$. The triples contained in relation $F$ represent the NoSQL data we wanted to query. The result is a pure SQL query which can be executed like any other query. When the query is executed relation $F$ retrieves the NoSQL data from the non-relational storage system and the data is automatically included in the query result as described by the user query.

A naive translation of the NoSQL query pattern to SQL causes significant overhead. We therefore propose some non-trivial and more advanced processing strategies to optimize the translation. These query processing strategies are mainly aimed at reducing the amount of triples shipped from the NoSQL database to the relational database. This is achieved by pushing down selection and projection conditions to the NoSQL database instead of communicating the entire NoSQL database in triple format. Furthermore, we propose a strategy to combine all triples in a single relation that is materialized by the relational database prior to query execution. This reduces the communication cost to a single retrieve operation, after which the SQL database is able to execute the query using this temporarily stored triple relation with all required NoSQL data.

This framework is constructed independent of the underlying SQL and NoSQL databases. Using relational algebra notation and not going into implementation details regarding triple relation $F$, we are able to provide a generically applicable theoretical framework to bridge SQL and NoSQL. We have thereby solved Task 1 and Task 2, the first two tasks we had to address to provide an appropriate solution to bridge SQL and NoSQL.

### 7.2 Summary of practical results

To cover Task 3 and Task 4, the second two tasks of our general problem, we have performed a thorough empirical analysis of the developed theoretical framework using a prototype implementation. In contrast to the theoretical framework, the prototype implementation obviously cannot be constructed in a database-oblivious way. We therefore have chosen to use PostgreSQL and MongoDB as the underlying relational and non-relational databases for the experiment. To incorporate the NoSQL data in triple format in the relational database, we used the Multicorn foreign data wrapper abstraction. This allowed us to create a foreign table in PostgreSQL that can be used like any other relation in PostgreSQL. Except when a foreign table is used in a query, the records are actually retrieved on the fly from the external data source. In our case, the MongoDB data is transformed to its triple representation and shipped to the relational database.

To investigate the prototype performance in more detail, we have constructed multiple datasets. Using different data sources to create the dataset, varying the amount of SQL records, the size of the NoSQL data, and the join probability between the relational and non-relational data, we generated 13 different datasets used for the empirical analysis. Likewise, we constructed an extensive set of query templates covering different scenarios. For four different flow classes we generated four query templates such that they cover all combinations of interesting properties in the NoSQL query pattern.

Using this experimental framework we compared different implementation versions of the prototype. Firstly, we analyzed the base implementation with only the selection pushdown strategy applied. To
investigate the effect of the projection pushdown we included this strategy in a second implementation which we can compare to the base implementation. Finally, we also applied the data reduction strategy using the materialized temporary relation to check if this indeed significantly reduces the amount of NoSQL data that is shipped to the SQL database and thereby improves the performance of the prototype.

The empirical analysis indicated that reducing the amount of triples shipped from NoSQL to SQL indeed results in a lower average query execution time. Firstly, the projection pushdown improves the performance of the prototype with a factor 1.9, on average almost halving the time required to get a query result. Moreover, the use temporary relation further reduced the total size of the data sent from MongoDB to PostgreSQL, which meant another performance improvement with a factor 18.8 compared to the second implementation.

When we increased the limit on the amount of NoSQL data, which we had set to get a practicable experiment to compare the optimizations to the base implementation, we noticed that the average query time increased as a result of the extra communication involved and the additional processing required to join this larger NoSQL triple relation. While feasible for our experiment, the performance of the final solution with an acceptable limit on the NoSQL data is not good enough to be applicable in a business environment where fast data retrieval is required. We did however notice that further improvement of the prototype can be achieved by optimizing the joins of the triple relation and having more control over the query plan and execution.

Summarizing, the theoretical analysis showed that we have implemented a prototype of the designed theoretical framework that successfully bridges the gap between SQL and NoSQL. However, there still is enough space for further improvement and additional research is required in order to obtain an industrial strength hybrid SQL-NoSQL solution.

### 7.3 Future work

The approach we proposed, specified, formalized, and empirically analyzed using a prototype is far from the ultimate solution to bridge SQL and NoSQL. Besides completely different approaches, even within the same solution direction there are plenty of possibilities to narrow the gap between SQL and NoSQL by improving the performance of the hybrid database to bridge both data storage worlds. This section provides some suggestions for possible follow-up research based on our solution.

Our prototype implementation can be further optimized by reducing the amount of triples that should be shipped from the NoSQL to the SQL database, as suggested in Section 7.3.1. Furthermore, Section 7.3.2 offers possible ways to more efficiently join the triples at the SQL side. On a bigger scale, we describe potential strategies to reduce the query time for more complex queries and sets of queries in Section 7.3.3. In Section 7.3.4 we discuss run-time optimizations that could be applied during query execution, in contrast to our techniques that are all applied prior to query execution. Finally, Section 7.3.5 focuses on the lack of a standard NoSQL query language and covers the implications a standard hybrid query language could have on our framework.

#### 7.3.1 Nested join reduction

A possible method to decrease the amount of data that has to be shipped from the NoSQL database to the SQL side is by further decreasing the number of triples required to reconstruct the NoSQL data. In our final implementations $I_c$ and $I_d$, the temporary table still contains triples that are only
used to connect different triples and are themselves not part of the NoSQL result. This is the case when a NoSQL query pattern contains a variable that is nested, while not all of its parents are used in the SQL part of the user query. In our final prototype implementation, additional triples are required to ensure that the nested NoSQL data is correctly reconstructed.

Especially for deeply nested data, this can potentially cause that a significant portion of triples is not actually needed by the relational database. Consider the following NoSQL query pattern:

\[(k_1 : (k_2 : (\ldots k_n : (k : ?v) \ldots)))\]

There are \(n\) nesting levels, with keys \(k_1, k_2, \ldots, k_n\), until eventually we have a variable \(?v\) for which we want to bind values for key \(k\). Although only the variable binding for \(?v\) is of interest, in our final implementation all \(n\) triples with nesting information are available in relation \(F\) in the SQL database as well. Not only are the unnecessary triples shipped to the relational database, they are also all joined in order to mimic the NoSQL structure before eventually only using the \(?v\) value in the rest of the SQL query.

In this case it is easy to see that the \(n\) triples with nesting information are not required by the SQL database. But in general, when more complicated nesting structures and additional variables in alternative paths of one of the ancestors are used, it is not trivial which triples are required and which not.

However, investigating this opportunity to reduce the amount of data that has to be communicated to the relational database would certainly be worthwhile. Reducing the communication and, in this case, thereby also the number of joins could significantly improve the performance of the hybrid database for nested data.

### 7.3.2 Tuple reconstruction

Because we first transform the NoSQL data to triples to have a structured triple relation \(F\) at the SQL side, the NoSQL data is first split into different key-value pairs and later reconstructed by correctly joining the triples in the relational database. The flexibility these triples offer is good for a generic NoSQL data inclusion in SQL and furthermore allows the NoSQL data to be represented as a relational table. However, an obvious disadvantage is that the data has to be reconstructed via a series of joins.

The temporary relation \(F\) contains all triples required to achieve this reconstruction. When a large amount of NoSQL data, in terms of triples, must be communicated to the relational database, this results in a giant triple relation. The self joins required to reconstruct the NoSQL data can then have a huge impact on the performance of the entire query.

One of the benefits the temporary table offers, is that its possibilities are almost similar to ‘normal’ relations. For example, this provides the possibility to add indexes to \(F\) that might speed up the self joins. Candidate attributes would be the \(id\), which is heavily used during the series of joins. Because a NoSQL query pattern most likely declares many keys and searches for corresponding variable bindings for their values, the \(key\) attribute is another possible indexing candidate.

Moreover, we can also use the information that the self joins on the \(F\) relation combines triples that originate from the same NoSQL data entity. For triples at the same NoSQL query pattern nesting level, this can be directly derived from the fact that the join conditions states that the \(id\) values
7.3. FUTURE WORK

should be the same. Nested data from the same NoSQL entity however, has a separate id value which is linked via the value of another triple.

Despite the different id value, these nested triples are at some point joined to the triples at another nesting level. It can therefore be beneficial to provide this information to the SQL database, such that the query planner can take this into account. This can be implemented by adding an extra attribute to relation F that, in the transformation from NoSQL data to the triple representation, is filled with the same value for each triple from the same NoSQL data entity. Often the NoSQL database will have a data identifier available that can be used for this purpose. Otherwise, a unique identifier per data entity can be determined during the transformation to triples.

This additional attribute of F could then be used in an additional join condition added to each self join on relation F. Say we name the new attribute x, then we add the condition \( F_1.x = F_2.x \) if we want to join \( F_1 \) and \( F_2 \). Now, regardless of whether \( F_1 \) and \( F_2 \) contain data at the same nesting level in the NoSQL query pattern, the x values should be the same. Adding an index on this new attribute could increase the query performance, as the data can now also be ordered using this field to, for example, allow a merge join on this attribute.

7.3.3 Triple relation reuse

So far, we have only considered user queries with a single NoSQL query pattern in it. Though our solution is constructed to support multiple patterns in a single query, this functionality is not used in the empirical analysis and not explicitly discussed in this report. Further investigation of the possibilities and the performance impact when using more than one NoSQL query pattern in a single user query could provide useful insights.

User queries covering multiple triple relations also form a case where the overall performance of the final query can be optimized. Particularly when the same NoSQL database is used more than once, it might be possible to combine the triples in a single triple relation to minimize the overhead introduced by connecting to an external data source. Furthermore, returned triples might be reused by each NoSQL query pattern, which can also reduce the total amount of triples that has to be shipped from NoSQL.

Reusing F or a subset of its triples introduces some new problems that have to be solved. We have to distinguish the triples per NoSQL query pattern in order to correctly reconstruct the NoSQL data for each pattern. Achieving this is more complicated if triples can be reused by other patterns as well, since it then no longer belongs to a single NoSQL query pattern.

The general idea of using triples for more than one NoSQL query pattern can be taken a step further. If a set of queries that is executed sequentially contains similar queries, the temporary relation generated for one query can possibly be reused by another query. Moreover, instead of optimizing a single user query we can then also look at improving the performance for a given set of queries. That is, combining the triples required for multiple user queries in a single temporary relations can improve the performance for the entire set of queries if this relation can be used by a multiple queries.

Note that such multi-query optimizations are closely related to other research areas like OLAP\(^1\) view materialization. For large amounts of multi-dimensional data, determining which views are useful to materialize in order to improve the overall performance for OLAP tasks is essentially the same as determining what is the optimal set of triple relations to materialize given a set of user queries.

\(^1\)http://www.olapcouncil.org/
For both possible future research ideas, optimizing for more than one NoSQL query pattern per user query and multi-query optimization, it is important to take into account that materializing triple relations means that we have to sacrifice NoSQL data consistency. But, as for OLAP, there are situations for which this is not a problem and in these cases applying this type of optimization could reduce the overall query time and thereby improve the performance of the application.

### 7.3.4 Dynamic run-time query evaluation strategies

The query processing strategies described in Chapter 4 and implemented in our prototype all focus on the translation from user query to pure SQL. The user query is translated prior to query execution, and all strategies treated in this report aim at optimizing this static translation. As the empirical analysis showed, optimizing the translation can significantly improve the query performance by reducing the amount of triples required to reconstruct the NoSQL data on the relational database side.

After the translation however, the query is executed like a normal SQL query. During query evaluation, it is possible that additional constraints on the NoSQL data can be derived. For example, selections on the relational data could restrict the set of SQL records in the query result. This information could be used when accessing the triple relation $F$ to join NoSQL data to the SQL result. Triples describing data that will not be joined to the SQL data, because it has been filtered out already, do not have to be shipped from NoSQL to SQL. This additional constraint, which is dynamically derived during query execution, could thus be used to reduce the amount of NoSQL triples that have to be communicated between both underlying databases.

Similarly, other information that could reduce the amount of NoSQL entities that has to be sent to the relational database can become available during run-time query evaluation. Pushing these additional conditions down to the NoSQL database during query execution improves the overall performance of the prototype. This type of dynamic run-time strategies are a different approach at optimizing the prototype performance and are not covered in this report. Further research to determine which additional constraints we can derive at run-time and how we send these new conditions to the NoSQL database prior to data communication is required to practically apply this suggestion.

Building further on this idea, another possibility is to introduce query rewrite rules the query planner can apply to better reckon with the fact that the triple relation $F$ imports external data when it is accessed. Rules to ensure that as much selection and projection conditions as possible can be applied on the NoSQL data reduce the amount of triples shipped to the relational database.

Furthermore, usage of attributes from relation $F$ can be postponed as long as possible in the query plan. This extends the run-time period in which additional constraints can be dynamically derived and pushed down to the NoSQL database. In other words, rewrite rules can ensure that the maximum set of constraints on the triple relation are constructed and applied before the NoSQL data is communicated to the SQL database.

### 7.3.5 NoSQL query language standardization

The query language proposed in Section 3.2 includes a NoSQL query pattern in a normal SQL statement. This way, the user is able to read and combine data from an SQL and a NoSQL source in a single query. However, a disadvantage of this query syntax is the clear separation between the SQL and NoSQL part. Although this is convenient if we want to process a query, it might still feel like two separate query languages merged together in a single statement.
Since our main goal is to bridge the gap between SQL and NoSQL, it would make sense to have a query language that is more hybrid than including one query language in the other. Our query language reminds the user of the underlying separated database systems, which was exactly what we wanted to prevent. A direction for future work could therefore be the development of a more uniform, hybrid query language that covers both SQL and NoSQL.

For our solution this would imply that we have to adjust our implementation to conform to the query language specification. This implies another translation from a NoSQL query pattern to SQL and possibly modifications in the transformation of NoSQL data to triples as well. Our framework however, where NoSQL data is included in a relational database and an extended SQL-like query language that allows developers to query the NoSQL data, still is a solid basis for any hybrid SQL-NoSQL database regardless of the query language.

During the course of our investigations, the authors of SQLite and CouchDB have proposed UnQL as an attempt to create a hybrid SQL-NoSQL query language. Like our own proposed query language, UnQL is based on SQL with an extension to query NoSQL data. In fact, UnQL is a superset of SQL and treats normal SQL relations as a special, heavily structured data type. Similar to our query language, UnQL uses a JSON-like syntax for NoSQL data. An example of UnQL, where \texttt{abc} is NoSQL data is given in Listing 7.1.

\begin{lstlisting}[language=sql]
SELECT 
  { 
    x:abc.type, 
    y:abc.content.x, 
    z:abc.content.x + 50 
  } 
FROM abc 
WHERE abc.type == 'message'
\end{lstlisting}

\textbf{Listing 7.1: Query in UnQL}

Instead of inserting a NoSQL query pattern in an SQL query, UnQL uses attributes formatted as paths to refer to NoSQL data. In the \texttt{select} part of the query these paths are prefixed with the attribute name used in the query result. This syntax is also convenient for other types of queries, like update and delete statements. In contrast to our query language, the selection conditions are completely separated from the \texttt{from} part of the SQL query. This looks cleaner, but the NoSQL query pattern in our proposal offers a shorter notation for more complicated selection conditions like checking if a value exists or not.

The brackets indicate that a projection on NoSQL data is performed, and similar to a NoSQL query pattern this distinguishes the NoSQL part of the query. The brackets are more subtle however, and moreover optional. A disadvantage of UnQL is the repetition of the paths that is required if multiple values from a nested part of the NoSQL data are desired. This can be more compactly denoted in the syntax we have proposed and used in our prototype.

Summarizing, UnQL has some useful aspects that could certainly be adopted by our query language, mainly because UnQL is a more hybrid notation compared to our NoSQL query pattern in SQL solution. Vice versa, our query language provides a more compact way to treat nested data. A standard NoSQL query language specification that combines the strengths of available ideas and offers a generic way to query SQL and NoSQL data at the same time would help to further bridge the gap between SQL and NoSQL data storage in the future.

\footnote{\url{http://www.unqspec.org/display/UnQL/Home}}
In this appendix we list the query templates used for the empirical analysis as discussed in Section 5.4. The query templates contain placeholders that are filled with constant values to generate an actual query that is translated and executed in the experiment. In Section A.1 the templates for the product datasets are given, and Section A.2 displays the templates for the *Twitter* dataset.

### A.1 Product queries

```sql
SELECT p.t AS type, p.p AS price, p.c AS copies FROM NoSQL( type: ?t, versions: ( 1: ( price: ?p, copies: ?c ) ) ) AS p WHERE p.c >= %3\$d AND p.p < %4\$d
```

**Listing A.1**: Product query template for $F_i$ and $Q_{i1}$

```sql
```

**Listing A.2**: Product query template for $F_i$ and $Q_{i2}$
APPENDIX A. QUERY TEMPLATES

Listing A.2: Product query template for $F_1$ and $Q_2$

```sql
SELECT p.t AS type, p.p AS price
FROM NoSQL(type: ?t, versions: (1: (price: ?p, copies: ?c)) ) AS p
WHERE p.p < %4$d
```

Listing A.3: Product query template for $F_1$ and $Q_3$

```sql
SELECT p.t AS type, p.p AS price, p.b AS booking_deadline
WHERE p.p < %4$d
```

Listing A.4: Product query template for $F_1$ and $Q_4$

```sql
SELECT p.i AS title, p.t AS type, p.p AS price, p.c AS copies, t.name
FROM NoSQL(title_id: ?i, type: ?t, versions: (1: (price: ?p, copies: ?c)) ) AS p,
titles AS t
```

74
WHERE
    p.i = t.id AND
    p.c >= %3\$d AND
    p.p < %4\$d

Listing A.5: Product query template for $F_{ii}$ and $Q_1$

```
SELECT
    p.i AS title,
    p.t AS type,
    p.l AS location,
    p.p AS price,
    p.c AS copies,
    p.b AS booking_deadline,
    t.name
FROM
    NoSQL(
        title_id: ?i, type: ?t, location: ?l, versions: (
            1: (
                price: ?p,
                copies: ?c,
                booking_deadline: ?b
            )
        )
    ) AS p,
    titles AS t
WHERE
    p.i = t.id AND
    p.c >= %3\$d AND
    p.p < %4\$d
```

Listing A.6: Product query template for $F_{ii}$ and $Q_2$

```
SELECT
    p.t AS type,
    p.p AS price,
    t.name
FROM
    NoSQL(
        title_id: ?i, type: ?t, versions: (  
            1: (  
                price: ?p,
                copies: ?c
            )
        )
    ) AS p,
    titles AS t
WHERE
    p.i = t.id AND
    p.p < %4\$d
```

Listing A.7: Product query template for $F_{ii}$ and $Q_3$

```
SELECT
    p.t AS type,
    p.p AS price,
    p.b AS booking_deadline,
    t.name
FROM
    NoSQL(
        title_id: ?i, type: ?t, location: ?l, versions: (  
```
APPENDIX A. QUERIES TEMPLATES

1: (  
   price: ?p,  
   copies: ?c,  
   booking_deadline: ?b  
 )  
) AS p,  
titles AS t  
WHERE  
p.i = t.id AND  
p.p < %4\$d

Listing A.8: Product query template for $F_{ii}$ and $Q_{1}$

1
SELECT  
c.name,  
p.t AS type,  
p.j AS concern,  
p.p AS price,  
p.c AS copies  
FROM  
concerns AS c,  
NoSQL(  
   type: ?t, versions: (  
      1: (  
         concern_id: ?j,  
         price: ?p,  
         copies: ?c  
      )  
   )  
) AS p  
WHERE  
c.id = p.j AND  
c.id < %2\$d AND  
p.c >= %3\$d AND  
p.p < %4\$d

Listing A.9: Product query template for $F_{iii}$ and $Q_{1}$

1
SELECT  
c.name,  
p.t AS type,  
p.l AS location,  
p.j AS concern,  
p.p AS price,  
p.c AS copies,  
p.b AS booking_deadline  
FROM  
concerns AS c,  
NoSQL(  
   type: ?t, location: ?l, versions: (  
      1: (  
         concern_id: ?j,  
         price: ?p,  
         copies: ?c,  
         booking_deadline: ?b  
      )  
   )  
) AS p  
WHERE  
c.id = p.j AND
A.1. PRODUCT QUERIES

Listing A.10: Product query template for \( F_{iii} \) and \( Q_2 \)

```sql
SELECT
c.name,
p.t AS type,
p.p AS price
FROM
c.concerns AS c,
NoSQL(
)
) AS p
WHERE
c.id = p.j AND
c.id < %2$d AND
p.p < %4$d
```

Listing A.11: Product query template for \( F_{iii} \) and \( Q_3 \)

```sql
SELECT
c.name,
p.t AS type,
p.p AS price,
p.b AS booking_deadline
FROM
c.concerns AS c,
NoSQL(
)
) AS p
WHERE
    c.id = p.j AND
c.id < %2$d AND
p.p < %4$d
```

Listing A.12: Product query template for \( F_{iii} \) and \( Q_4 \)

```sql
SELECT
t.name AS title_name,
p.i AS title,
p.t AS type,
p.j AS concern,
p.p AS price,
p.c AS copies,
c.name AS concern_name
FROM
```
APPENDIX A. QUERY TEMPLATES

Listing A.13: Product query template for $F_{iw}$ and $Q_1$

```sql
SELECT
  t.name AS title_name,
  p.i AS title,
  p.t AS type,
  p.l AS location,
  p.j AS concern,
  p.p AS price,
  p.c AS copies,
  p.b AS booking_deadline,
  c.name AS concern_name
FROM
  titles AS t,
  NoSQL(
    title_id: ?i, type: ?t, location: ?l, versions: (1:
      concern_id: ?j,
      price: ?p,
      copies: ?c,
      booking_deadline: ?b
    )
  ) AS p,
  concerns AS c
WHERE
  t.id = p.i AND
  t.id < %1$d AND
  p.j = c.id AND
  p.c >= %3$d AND
  p.p < %4$d
```

Listing A.14: Product query template for $F_{iw}$ and $Q_2$

```sql
SELECT
  t.name AS title_name,
  p.t AS type,
  p.p AS price,
  c.name AS concern_name
FROM
  titles AS t,
  NoSQL(
    title_id: ?i, type: ?t, location: ?, versions: (1:
      concern_id: ?j,
      price: ?p,
      copies: ?c,
      booking_deadline: ?b
    )
  ) AS p,
  concerns AS c
WHERE
  t.id = p.i AND
  t.id < %1$d AND
  p.j = c.id AND
  p.c >= %3$d AND
  p.p < %4$d
```

78
A.2 Tweet queries

SELECT
    t.c AS created_at,
    t.t AS text
FROM
    NoSQL(
    ) AS t
WHERE
    t.l = %1$d

Listing A.17: Tweet query template for $F_i$ and $Q_4$
APPENDIX A. QUERY TEMPLATES

SELECT
  t.c AS created_at,
  t.t AS text,
  t.u AS url,
  t.m AS user_mention_id
FROM
  NoSQL(  
    created_at: ?c, iso_language_code: ?l, text: ?t, entities: (  
      urls: (  
        0: (  
          url: ?u  
        )  
      ),  
      user_mentions: (  
        0: ?m  
      )  
    )  
  ) AS t
WHERE
  t.l = %1$d

Listing A.18: Tweet query template for $F_i$ and $Q_2$

SELECT
  t.t AS text
FROM
  NoSQL(  
  ) AS t
WHERE
  t.l = %1$d

Listing A.19: Tweet query template for $F_i$ and $Q_3$

SELECT
  t.t AS text,
  t.m AS user_mention_id
FROM
  NoSQL(  
    created_at: ?c, iso_language_code: ?l, text: ?t, entities: (  
      urls: (  
        0: (  
          url: ?u  
        )  
      ),  
      user_mentions: (  
        0: ?m  
      )  
    )  
  ) AS t
WHERE
  t.l = %1$d

Listing A.20: Tweet query template for $F_i$ and $Q_4$

SELECT
  t.c AS created_at,
  t.t AS text,
  u.name,
  u.screen_name
A.2. TWEET QUERIES

FROM

NoSQL(
) AS t,
users AS u
WHERE

t.i = u.id AND
t.l = %1$d

Listing A.21: Tweet query template for \( F_{ii} \) and \( Q_1 \)

SELECT
t.c AS created_at,
t.t AS text,
t.u AS url,
t.m AS user_mention_id,
u.name,
u.screen_name
FROM

NoSQL(
    user_mentions: (0: (?m))
) AS t,
users AS u
WHERE
t.i = u.id AND
t.l = %1$d

Listing A.22: Tweet query template for \( F_{ii} \) and \( Q_2 \)

SELECT
t.t AS text,
u.name,
u.screen_name
FROM

NoSQL(
) AS t,
users AS u
WHERE
t.i = u.id AND
t.l = %1$d

Listing A.23: Tweet query template for \( F_{ii} \) and \( Q_3 \)
APPENDIX A. QUERY TEMPLATES

Listing A.24: Tweet query template for $F_{ii}$ and $Q_4$

```sql
SELECT u.name, u.screen_name, t.c AS created_at, t.t AS text
FROM users AS u,
NoSQL(
) AS t
WHERE u.id = t.i AND u.id < %2\$d AND t.l = %1\$d
```

Listing A.25: Tweet query template for $F_{iii}$ and $Q_1$

```sql
SELECT u.name, u.screen_name, t.c AS created_at, t.t AS text, t.u AS url, t.m AS user_mention_id
FROM users AS u,
NoSQL(
        urls: (
            0: (url: ?u)
        ),
        user_mentions: (0: ?m)
    )
) AS t
WHERE u.id = t.i AND u.id < %2\$d AND t.l = %1\$d
```

Listing A.26: Tweet query template for $F_{iii}$ and $Q_2$

```sql
SELECT u.name, u.screen_name, t.c AS created_at, t.t AS text
FROM users AS u,
NoSQL(
        urls: (
            0: (url: ?u)
        ),
        user_mentions: (0: ?m)
    )
) AS t
WHERE u.id = t.i AND u.id < %2\$d AND t.l = %1\$d
```
A.2. TWEET QUERIES

Listing A.27: Tweet query template for Q_3

Listing A.28: Tweet query template for Q_4

Listing A.29: Tweet query template for Q_1
APPENDIX A. QUERY TEMPLATES

Listing A.30: Tweet query template for $F_{iv}$ and $Q_2$

```sql
SELECT
    u.name,
    u.screen_name,
    t.c AS created_at,
    t.t AS text,
    t.u AS url,
    t.m AS user_mention_id,
    l.iso_language_code
FROM
    users AS u,
    NoSQL(?
            urls: (?
                url: ?u
            ),
            user_mentions: (?
                ?m
            )
        )
    ) AS t,
    iso_language_codes AS l
WHERE
    u.id = t.i AND
    u.id < %2$d AND
    t.l = l.id AND
    t.l = %1$d
```

Listing A.31: Tweet query template for $F_{iv}$ and $Q_1$

```sql
SELECT
    u.name,
    u.screen_name,
    t.t AS text,
    t.m AS user_mention_id,
    l.iso_language_code
FROM
    users AS u,
    NoSQL(?
            urls: (?
                url: ?u
            ),
            user_mentions: (?
                m
            )
        )
    ) AS t,
    iso_language_codes AS l
WHERE
    u.id = t.i AND
    u.id < 2\$d AND
    t.m = l.id AND
    t.m = %1\$d
```

84
Listing A.32: Tweet query template for $F_{iv}$ and $Q_A$
This appendix shows the foreign data wrapper implementation we used to implement a prototype of the developed theoretical framework. As explained in Section 5.5, we use MULTICORN for this implementation. The foreign data wrapper is therefore written in PYTHON. Note that this version of the code includes selection and projection pushdown support. The version without projection pushdown, as used for implementation $\mathcal{I}_a$, is obtained via a trivial modification to exclude the use of condition.['nosql_projection'].

```python
# Required libraries
from multicorn import ForeignDataWrapper
from pymongo import Connection
import json

class MongoForeignDataWrapper(ForeignDataWrapper):

    # Initialize FDW
    def __init__(self, options, columns):
        super(MongoForeignDataWrapper, self).__init__(options, columns)

    # Database and collection name are provided by Multicorn via the options parameter
    self.limit = 25000
    self.connection = Connection()
    self.database = self.connection[options.get('database', 'test')]
    self.collection = self.database[options.get('collection', None)]

    # Retrieve data
    def execute(self, quals, columns):
        self.condition = {}

        # Quals are selection conditions
        for q in quals:
            if q.operator == '=':
                self.condition[q.field_name] = q.value

        # Retrieve data from MongoDB using the pushed down selection and projection
        for d in self.collection.find(json.loads(self.condition['nosql_query']), json.loads(self.condition['nosql_projection'])).limit(self.limit):
            for t in self.process_dict(['_id', str(d['_id'])], d):
                yield t
```
# Function psi

```python
def process_dict(self, i, d):
    return reduce(lambda x, y: x + y, map(lambda (k, v): self.process(i, str(k), v), d.iteritems()), [])
```

# Function psi for lists

```python
def process_list(self, i, l):
    return reduce(lambda x, y: x + y, map(lambda (k, v): self.process(i, str(k), v), enumerate(l)), [])
```

# Function phi

```python
def process(self, i, k, v):
    # Set default attribute values
    row = {
        'id': '_' .join(i),
        'key': k,
        'nosql_query': self.condition['nosql_query'],
        'nosql_projection': self.condition['nosql_projection']
    }

    # Generate new unique id
    # i uniquely identifies the current set && k is unique for this i
    # => i + [k] is unique
    j = i + [k]

    # Case distinction on value type
    if type(v) is dict:
        # Nested set
        row['value'] = '_' .join(j)
        return [row] + self.process_dict(j, v)
    elif type(v) is list:
        # List => nested set
        row['value'] = '_' .join(j)
        return [row] + self.process_list(j, v)
    else:
        # No nesting
        row['value'] = v
        return [row]
```

Listing B.1: Foreign data wrapper in Multicorn


