Doctoral Thesis

Scalable systems for data analytics and integration

Author[s]:
Hentschel, Martin

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007328025

Rights / License:
In Copyright - Non-Commercial Use Permitted
SCALABLE SYSTEMS FOR DATA ANALYTICS AND INTEGRATION

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

MARTIN HENTSCHEL

Master of Science ETH in Computer Science, ETH Zurich

January 4, 1982

citizen of Germany

accepted on the recommendation of

Prof. Dr. Donald Kossmann, examiner
Dr. Laura Haas, co-examiner
Prof. Dr. Guido Moerkotte, co-examiner
Prof. Dr. Nesime Tatbul, co-examiner

2012
Abstract

The Internet allows people and businesses to communicate and collaborate at an unprecedented scale. As a result, several new trends are emerging in the world of information. Two of these are: (1) more and more information is shared in real time, such as text messages and pictures in social networks like Twitter and Facebook; and (2) interchanged information has increasingly different formats, or schemas. For example, medical data is interchanged in different schemas depending on the hospital or insurance company. Information systems that support these trends must, therefore, process and analyze data in real time and integrate data in many different schemas. Since both trends are ongoing, there is a need for systems that scale in these dimensions.

Today’s information systems, however, scale insufficiently. As we will show, current systems for real-time analytics sacrifice important factors such as data consistency; current approaches to data integration do not scale with the number of schemas of interchanged messages. Therefore, we address the following two research questions in this thesis: (1) how to build a system for real-time analytics that scales with the number of input messages and satisfies important criteria such as strong consistency and fault tolerance; and (2) how to build a system for data integration that scales with the number of schemas of interchanged messages. Both questions are studied in individual parts of this work. An outlook on how to combine the systems of both parts is given at the end of the thesis.

In the first part of this thesis we present Limmat, a scalable system for real-time data analytics, and make the following contributions. In a first step, we analyze different approaches for distributing information in social networks, and present a new approach that utilizes system resources better than existing approaches. Based on these findings, we study the design space of systems for real-time analytics, and develop Limmat. Limmat analyzes data in real time, scales with the number of input messages, and guarantees strong consistency and fault tolerance. The support of strong consistency sets Limmat apart from other real-time analytics systems such as Google Percolator and Twitter Storm. Finally, we present an application that shows how Limmat can be employed in practice.
In the second part of this thesis we present *Mapping Data to Queries (MDQ)*, a new system for data integration. MDQ differs from existing approaches to data integration by annotating data with additional access paths at the runtime of a query. Integrating data at the runtime of a query offers great potential for optimization, which allows MDQ to scale with the number of schemas of interchanged messages. MDQ outperforms existing approaches by orders of magnitudes in extreme cases. As further contributions, we present two new concepts in the field of data integration: *holistic data integration* and *integration independence*. Holistic data integration allows both schema-level and data-level integration within a single system. Integration independence decouples applications from when, where, and how the integration takes place. We show how MDQ supports these concepts. Finally, we present an application that builds on top of MDQ.

In summary, the contributions of this thesis support two ongoing trends in the world of information. Limmat analyzes data in real time, while scaling with the number of messages and ensuring strong consistency. MDQ integrates data in increasingly different schemas. A combination of these systems, which supports both trends simultaneously, is described at the end of this thesis. However, an experimental study of such a combined system is part of future work.
Das Internet ermöglicht Personen und Unternehmen in einer nie dagewesenen Wei-
se miteinander zu kommunizieren und zusammen zu arbeiten. Dabei sind mehrere
neue Trends in der heutigen IT-Welt zu beobachten. Zwei davon sind: 1. Mehr
und mehr Informationen werden in Echtzeit ausgetauscht, wie zum Beispiel Kurz-
nachrichten und Bilder in sozialen Netzwerken wie Twitter und Facebook. 2. Aus-
getauschte Daten haben immer häufiger verschiedene Formate bzw. Schemas, wie
zum Beispiel Patientendaten im Gesundheitswesen, die je nach Krankenhaus und
Versicherung in verschiedenen Schemas vorhanden sind. Informationssysteme, die
diese Trends unterstützen, müssen daher Daten in Echtzeit verarbeiten und analy-
sieren können, und Daten mit vielen verschiedenen Schemas integrieren. Da beide
Trends anhalten, sind skalierende Systeme erforderlich.

Heutige Informationssysteme skalieren jedoch nur unzureichend. Wie wir zeigen,
verletzen derzeitige Systeme zur Echtzeitanalyse wichtige Eigenschaften wie z.B.
Datenkonsistenz. Derzeitige Systeme zur Integration von Daten können nicht mit
der Anzahl der Schemas von ausgetauschten Daten skalieren. Deswegen beschäf-
tigen wir uns in dieser Arbeit mit den folgenden zwei Forschungsfragen: 1. Wie
baut man ein System zur Echtzeitanalyse, das mit den Eingabedaten skaliert und
gleichzeitig wichtige Eigenschaften wie Datenkonsistenz erfüllt. 2. Wie baut man
ein System zur Datenintegration, das mit der Anzahl der Schemas von ausge-
tauschten Daten skaliert. Beide Fragen werden in unterschiedlichen Teilen dieser
Arbeit beantwortet. Ein Ausblick, wie man beide Lösungen verbinden kann, wird
am Ende der Arbeit gegeben.

Im ersten Teil dieser Arbeit stellen wir Limmat vor, ein System zur Echtzeitanaly-
se. Dabei kommen wir zu folgenden Ergebnissen. In einer zu Limmat vorausgehend-
en Studie untersuchen wir verschiedene Ansätze zur Verteilung von Informationen
in sozialen Netzwerken. Wir präsentieren einen neuen Ansatz, der Systemressour-
cen besser als bisherige Ansätze ausnutzt. Darauf aufbauend untersuchen wir An-
forderungen von Systemen zur Echtzeitanalyse, und entwickeln Limmat. Limmat
analysiert Daten in Echtzeit, skaliert mit den Eingabedaten und gewährleistet Da-
tenkonsistenz und Fehlertoleranz. Die Gewährleistung von Datenkonsistenz hebt


Acknowledgments

First of all, I would like to thank my advisor, Prof. Donald Kossmann. I first met Donald in 2005, when I was a student in the Introduction to Databases course at ETH Zurich. He was one of the professors who could motivate the students to understand the topic a tiny bit more, simply by being excited about it himself. And to this day, Donald is one of the most passionate people I have ever met. He is, without a doubt, passionate about everything: doing research and writing research papers; turning research into business; discussing crazy, new ideas; and—skiing. I am truly thankful for his support and constant motivation throughout my time at ETH Zurich.

I would like to thank the members of my examination committee, Laura Haas, Prof. Guido Moerkotte, and Prof. Nesime Tatbul. In particular, I want to thank Prof. Guido Moerkotte for his extensive feedback on this thesis, which helped to improve the work significantly. I also thank Prof. David Basin for being available to chair my examination committee.

There are several people to whom I am grateful for their excellent help in different stages of Mapping Data to Queries. With Tim Kraska and Jonas Rutishauser, I implemented, tested, and optimized the first prototype of MDQ. With Laura Haas and Prof. Renée J. Miller, I discussed and implemented many ideas that took MDQ to the next level; their help and support with MDQ over the past years was really great. I also thank Sukriti Ramesh for her help in analyzing and benchmarking the storage component of MDQ, and Prof. Peter Fischer for his help in all things XML and XQuery related.

I am happy to have met Maria Grineva and Maxim Grinev during the course of my studies. Together with both of them, I developed Limmat. Their constant flow of new ideas and their excitement about new technologies both in research and in industry is fascinating. I thank Maria and Maxim for the wonderful time that we had together during the one and a half years when they were in Zurich.

I would also like to thank my colleagues in the Systems Group at ETH Zurich for being so wonderfully pleasant and friendly to work with; especially Simonetta
Zysset, Eva Ruiz Muller, and Jena Bakula who always helped with organizational issues; Irina Botan, Qin Yin, Kyumars Sheykh Esmaibli, and Philipp Unterbrunner with whom it was very enjoyable to share the office with; and all friends in the Systems Group for fantastic memories both at work and outside of work.

Finally, I would like to thank my parents, Ingrid and Thomas, and my sister, Maria, for their continuous support and motivation.
# Contents

## 1 Introduction

1.1 Motivation ........................................... 1
1.2 Research Questions ................................. 3
1.3 Contributions of this Thesis ....................... 4
1.4 Structure of this Thesis ............................. 5

## 2 Background

2.1 Key-Value Stores ..................................... 7
2.2 MapReduce ............................................ 10
2.3 XML and Semi-structured Data ....................... 13
2.4 XQuery ................................................. 17
2.5 YFilter ................................................ 18

## 3 Building Data Flows Using Key-Value Stores

3.1 Introduction .......................................... 23
3.2 Running Example ..................................... 27
3.3 Synchronous Approach ............................... 27
   3.3.1 Programming and Execution Models ............ 29
   3.3.2 Failure Model .................................... 29
   3.3.3 Scalability ....................................... 29
3.4 External Queue Approach ......................... 30
   3.4.1 Programming Model ............................. 30
   3.4.2 Execution Model ................................. 30
3.4.3 Failure Model ........................................... 31
3.4.4 Scalability .............................................. 32

3.5 Integrated Approach ...................................... 32
3.5.1 Programming Model .................................... 35
3.5.2 Execution Model ....................................... 35
3.5.3 Failure Model ........................................... 38
3.5.4 Scalability .............................................. 40

3.6 Experiments and Results .................................. 40
3.6.1 Saturating Workload ................................ 42
3.6.2 80% Workload ....................................... 43
3.6.3 Bursty Workload ..................................... 44
3.6.4 Fault Tolerance ...................................... 44
3.6.5 Scalability .............................................. 45

3.7 Related Work .............................................. 48
3.8 Conclusion ................................................... 50

4 The Limmat System ........................................... 51
4.1 Introduction ............................................... 51
4.2 Assumptions .............................................. 54
4.3 Programming Model ...................................... 55
4.4 Design Space ............................................. 57
4.4.1 Fault Tolerance Requirement ........................ 57
4.4.2 Scalability Requirement .............................. 59
4.4.3 Strong Consistency Requirement ................. 63
4.4.4 Summary .............................................. 65
4.5 Related Work .............................................. 66
4.6 Limmat ...................................................... 70
4.6.1 Assumptions .......................................... 70
4.6.2 Additional Mechanisms .............................. 71
4.6.3 Architecture .......................................... 72
4.6.4 Execution of map/reduce' .......................... 73
Chapter 1

Introduction

1.1 Motivation

The recent years have brought tremendous change to the world of information. The spread of the Internet now allows people and businesses to connect and collaborate at an unprecedented scale. As a result, several new trends are emerging in the world of information. Two of these are: (1) more and more data is shared in real time, and (2) data is increasingly becoming more manifold and diverse. Both trends are still ongoing as we will show with the help of prominent examples.

More and more data is shared in real time. With the success of the Internet, now being ubiquitously available, we are witnessing a tremendous shift towards the sharing of information in real time. Ten years ago, people used personal computers to connect to the Internet. Personal computers were hard to carry around, and people only periodically sent and received messages and news. Today, people use mobile phones that are constantly connected to the Internet. Mobile phones can be carried everywhere, which allows people to send and receive messages, news, and updates anywhere at any time. Real-time information sharing has become a key feature for successful services such as social networks, suggesting that this trend will continue.

Twitter, for example, is a service for real-time information sharing. It is an online social networking service that allows its users, individuals and news agencies alike, to distribute updates and news to other users in real time. Founded only five years ago, Twitter now has over 100 million active users [92] and processes 2000 messages per second on average [90]. These numbers show that Twitter has grown impressively and—most likely—will continue to do so.

Data is increasingly becoming more manifold and diverse. The Internet allows
businesses across the world to collaborate, and interchange data. Yet, each business has its own data formats, called schemas, for important data. Somewhat counterintuitively, this diversity is not going away but will even increase. Businesses will never agree on the one schema to interchange data. However, there are standards that define conventions for interoperability across schemas. Such standards give businesses the flexibility to define their own schemas, while allowing them to interchange data and interoperate. The success of such standards for interoperability implies that this trend will also continue.

Health Level 7, for example, is a standard in the health care industry that defines conventions for interoperability across schemas of interchanged data. Health Level 7 gives each hospital and health insurance company the flexibility to define their own schemas. Introduced in 1987, Health Level 7 is currently present in 55 countries and used by more than 4000 businesses and organizations [59]. As more and more businesses adapt the Health Level 7 standard, we will see a further increase in the number of different schemas and, therefore, face increasingly diverse data.

Both examples, Twitter and Health Level 7, represent major players in their respective fields, illustrating today’s demands. Twitter processes 2000 messages per second; Health Level 7 is used by thousands of businesses, each possibly defining their own schemas. As pointed out, both trends will continue. More and more data will be shared in real time, and the diversity of data will continue to grow.

In this thesis, we study the underlying information systems that support these trends.

A system for real-time information sharing, such as in Twitter, needs to support the following tasks. First, it needs to forward information posted by a user to other users, which results in (propagational) data flows that need to be handled. Second, the system needs to analyze the data in real time. The analysis of data in real time is important to summarize, aggregate, and rank information that is shared by users. This is challenging because it requires data consistency and fault tolerance, and the system needs to scale with an increasing amount of data that is shared. We call a system that supports the second task a real-time analytics system.

A system that processes interchanged data in different schemas, such as Health Level 7 data, must be able to integrate these data. Typically, such systems are provided with mapping rules, which establish connections between different schemas. As discussed, in industries such as health care there are thousands of different businesses and organizations, and therefore hundreds or even thousands of different schemas. Thus, a data integration system must be able to scale with an
Figure 1.1: Challenges addressed in this thesis. Scaling with the number of input messages (a), and scaling with the number of schemas of input messages (b).

increasing number of schemas of interchanged data.

Today’s information systems, however, scale insufficiently. Current systems for real-time analytics sacrifice important factors such as strong consistency; current approaches to data integration do not scale with the numbers of the schemas of interchanged data. In particular, Twitter Storm [110], the real-time analytics system of Twitter, does not guarantee strong consistency in case of machine failures; Google Percolator [97], the real-time analytics system of Google, only implements weak consistency guarantees (i.e., at-most-once semantics). Current approaches to data integration do not scale with the number of schemas of interchanged data: they apply mapping rules eagerly and without any optimization; and thus, inherently, cannot scale with the number of schemas.

1.2 Research Questions

The goal of this thesis is to build scalable systems for data analytics and integration. Therefore, this thesis addresses the following two research questions:

The first question is: How does one build a system for real-time analytics that scales with the number of input messages? This challenge is sketched in Figure 1.1(a). In Figure 1.1(a), a system analyzes input messages (in real time) and
produces output messages. Thereby, the focus lies on scaling with the number of
input messages. This is challenging because a system for real-time analytics should
not only analyze data in real time, but also provide guarantees that are considered
state-of-the-art for modern analytics systems. These guarantees include strong
consistency and tolerance to machine failures.

The second question is: How does one build a system for data integration that
scales with the number of schemas of input messages? This challenge is sketched
in Figure 1.1(b). In Figure 1.1(b), the input messages have different schemas,
illustrated by the different patterns. Here, the focus lies on scaling with the num-
ber of schemas: the system should provide constant (good) performance for an
increasing number of schemas of input messages. This is challenging because all
current approaches (inherently) fail to do so.

1.3 Contributions of this Thesis

In studying these research questions, this thesis makes the following contributions:

1. We study in detail how to build propagational data flows, which support,
   for example, the forwarding of information in social networks in real time.
   Thereby, we highlight an essential drawback of the state-of-the-art approach:
   poor resource utilization. We present an alternative approach to building
data flows that tightly couples data flow execution and data storage. Our
approach utilizes resources more efficiently, while preserving all the key fea-
tures of the state-of-the-art approach, most notably good scalability.

2. Based on these results, we list the requirements and study the design space
   options for building a real-time analytics system. As a result of this study, we
   present Limmat. Limmat guarantees strong consistency, which sets it apart
   from similar systems such as Google Percolator and Twitter Storm. Despite
   the use of distributed transactions needed for strong consistency, Limmat
   scales with the number of nodes of the system, and, thus, can process an
   increasing number of input messages.

3. We present an application on top of Limmat that optimizes online newspa-
   pers using social media. It shows how Limmat can be employed in practice.

4. We present Mapping Data to Queries (MDQ), a scalable system to integrate
data. MDQ differs significantly from traditional approaches to data integra-
tion by integrating data at the latest possible point in time, at the runtime
of a query. This opens up a great potential for optimization. Consequently,
MDQ scales well with the number of schemas, and outperforms traditional approaches by orders of magnitude in extreme cases.

5. We introduce two new concepts in the field of data integration: holistic data integration and integration independence. Holistic data integration aims at performing schema-level and data-level integration tasks within the same integration engine. Integration independence aims at moving transparently between a materialized view and a virtual view of the integrated data. We show how MDQ supports these two concepts.

6. Finally, we pitch People People People, an application that integrates personal data from various online and offline data sources. People People People runs on top of MDQ showing how MDQ can be used in practice.

A combination of Limmat and MDQ results in a single system that scales in both dimensions: with the number of input messages and with the number of schemas of input messages. A practical example that has both many real-time messages and many schemas is intelligence services. Intelligence services analyze many messages in many different schemas in real time (e.g., filtering emails and messages in social networks for certain keywords) and, thus, would benefit from a combination of Limmat and MDQ.

1.4 Structure of this Thesis

The remainder of this thesis is divided into two parts. The first part studies propagational data flows and real-time analytics systems. The second part studies the field of data integration. In detail, this thesis is structured as follows:

Chapter 2 presents an overview of existing technologies on which this thesis builds.

Part 1 explores propagational data flows and real-time analytics systems. It consists of chapters 3, 4, and 5. Chapter 3 studies different approaches to building data flows, including our new approach. Chapter 4 presents Limmat, a system for real-time analytics, and compares Limmat to related systems from Google and Twitter. Chapter 5 highlights an application on top of Limmat, which optimizes online newspapers using social media.

Part 2 then explores the field of data integration and consists of chapters 6, 7, and 8. Chapter 6 presents Mapping Data to Queries (MDQ), our new system for data integration. Chapter 7 introduces the concepts of holistic data integration and integration independence, and explains how MDQ supports these concepts.
Chapter 8 presents the People People People application, which runs on top of MDQ.

Finally, Chapter 9 concludes this thesis. We give an outlook of future work, which includes how to combine the two systems, Limmat and MDQ, into a single system.
Chapter 2

Background

In this chapter we describe the underlying technologies this thesis builds on. The first part of this thesis builds on key-value stores and MapReduce. Key-value stores are a light-weight alternative to traditional database systems with a strong focus on scalability. MapReduce is a framework for data analytics, which became popular for its simple interface, fault tolerance, and scalability as well. The second part of this thesis builds on XML and related technologies XQuery and YFilter. XML is a popular format to interchange data. XQuery is a programming language to process XML data; while YFilter indexes specific XQuery expressions to quickly match XML documents. Next, we will explain each of these technologies in detail, and highlight their importance with respect to this thesis.

2.1 Key-Value Stores

Key-value stores are systems that store data in the form of key-value pairs. A key-value pair is a data representation where a value is assigned to a key. The key is a unique identifier that determines the value. The value can be any type of data; for example strings, images, or video files. An example of a simple key-value pair is 17="Anna". Here, the key is a unique person identifier and the value is a string. Key-value stores are able to store countless numbers of such key-value pairs, typically in the range of terabytes and beyond.

Key-value stores are designed to be highly scalable and available. Key-value stores scale with the number of machines in the system—typically called nodes. The more nodes are added to the system, the more data can be stored and the more operations (to store and retrieve data) can be performed per second. Or, in other words, adding more nodes to the system increases the capacity and the throughput
of the system. Furthermore, key-value stores are designed to be highly available. If some of the nodes of the key-value store are down, for example because of a faulty process or a local network outage, the key-value store as a whole continues to function. The system remains available even if some of its nodes are unavailable. These two properties, scalability and availability, have made key-value stores a popular choice to store large amounts of data.

Compared to traditional relational database systems however, key-value stores have a number of limitations. These limitations include no support for schemas, a simpler query interface, no support for secondary indexes, and no transactions:

- **No Schemas.** In relational databases, every table has a fixed schema, which defines the names and types of the table’s records. A record is an entry in the table (in other words, a row), which consists of multiple values. Key-value stores, on the other hand, have no support for schemas. Data is managed in the form of key-value pairs, without any type information. Some key-value stores, however, offer limited support for schemas by partitioning the value of a key-value pair into multiple smaller key-value pairs (a “row”) [27, 74] or allowing to store key-value pairs in different tables [74].

- **Simple Interface.** Relational databases provide an interface to query data using SQL, a powerful query language. Key-value stores provide a much simpler query interface, essentially allowing to read and write key-value pairs. Some key-value stores additionally allow to query ranges of data [27, 74].

- **No Secondary Indexes.** Relational databases support secondary indexes that allow to quickly retrieve records by values other than their key. For example, using secondary indexes, an employee record can be quickly accessed by name or department instead of only by employee id (the key). Key-value stores do not support secondary indexes. Key-value pairs can only be retrieved by key.

- **No Transactions.** Relational databases support transactions. A transaction is the combination of multiple database operations, which are executed as one unit without adverse effects from other transactions [67]. A set of properties ensures that transactions are executed reliably: atomicity, consistency, isolation, and durability (ACID). Key-value stores do not support transactions; multiple operations cannot be combined as one unit. In particular, multiple operations cannot be executed atomically [34, 74]. However, some key-value stores offer transactions on a single row [27].

Despite these limitations, many commercial and non-commercial key-value stores have been employed in practice. Well-known commercial key-value stores are, for
example, Amazon Dynamo [34] and Google Bigtable [27]. Non-commercial key-value stores include Apache Cassandra [74], BerkeleyDB [91], and memcached [82]. Most non-commercial key-value stores are even available for free, which led some researchers to the conclusion that this “cheap” price in comparison to traditional database systems is another important factor of the success of key-value stores.\(^1\)

The main working principle of key-value stores is the partitioning and replication of data. Data is partitioned across all nodes of the key-value store using consistent hashing [65]. In consistent hashing, the domain of a hash function is treated as a fixed circular space—a ring. Each node of the key-value store is responsible for a part of this ring to store data. Adding more nodes to the system shifts data and requests from loaded nodes to the new nodes, which in total increases the capacity and throughput of the system. The system scales incrementally with the number of nodes. To facilitate availability, data is replicated within the system. Each key-value pair is replicated to multiple nodes, which form a replication group. Typically, quorum-based protocols are used to replicate the data; only the majority of nodes of the replication group (the quorum) needs to be available to read and write data. In summary, partitioning of data enables scalability, while replication of data ensures availability.

Figure 2.1 shows the typical illustration of a key-value store in the literature. This figure is similar to the illustration in the Dynamo article [34, Fig. 2]. In Figure 2.1, the key-value store consists of five nodes, \(N_0\) to \(N_4\). The nodes are arranged in a ring, which is specified by the hash function. Node \(N_1\) is responsible for storing keys in the range \((N_0, N_1)\). The key-value pair \(k_1, v_1\), which falls in this range, will be stored at Node \(N_1\). Furthermore, \(k_1, v_1\) is replicated to nodes \(N_2\) and \(N_3\), which together with \(N_1\) form the replication group for this pair.

Key-value stores are important for this thesis because they serve as scalable and

\(^1\)Hector Garcia-Molina during a panel discussion at the 2010 VLDB conference [2].
available backend of our data analytics system. We will show how to extend key-value stores with mechanisms for data processing (i.e., queues and worker threads) to support data flows. Furthermore, we will extend key-value stores with mechanisms for consistency (i.e., transactions and the two-phase commit protocol) to support real-time data analytics. We will show that all these extensions never sacrifice the built-in scalability of key-value stores.

2.2 MapReduce

MapReduce is a popular tool for data analytics introduced by Google in 2004 [32]. Since then, MapReduce is successfully used in businesses such as Google and Facebook [33, 109], but it also sparked interest in the research community. Arguably, the reasons of the success of MapReduce are (1) its simple, yet powerful programming model; (2) the high scalability of its execution framework; and (3) the ability to tolerate machine failures. Unfortunately though, MapReduce shows poor data freshness and, thus, is not suited for data analytics in real time. In the following, we will first discuss the successful properties of MapReduce and then explain its shortcoming in detail.

Programming Model. The programming model of MapReduce is simple yet powerful at the same time. It can be applied to a variety of analytical tasks, ranging from simple counting to large-scale graph computations [33]. Figure 2.2 illustrates the MapReduce programming model. It simply consists of two functions, map and reduce. (A third function, combine, exists to optimize parts of the computation.) As shown in Figure 2.2, the map function takes as input a key-value pair and outputs a list of key-value pairs. The reduce function takes as input a key and a corresponding list of values, and outputs a potentially smaller list of values.

An example of how the programming model works is the simple word count application from the original paper of MapReduce [32]. The application counts the number of occurrences of each word in a document. The pseudo code using the

\[
\text{map} : \quad (k_1, v_1) \rightarrow \text{list}(k_2, v_2) \\
\text{reduce} : (k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)
\]

Figure 2.2: The MapReduce programming model.
MapReduce programming model (with proper types of input values) looks like this:

```java
map(String key, String value):
    for each word w in value:
        emit (w, 1);

reduce(String key, List<int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    emit (key, sum);
```

The map function emits (word, 1) for each word in the input. The reduce function sums up the list of counts for a particular word, and emits a single key-value pair containing the word and its final sum.

The definition of the reduce function of Figure 2.2 can be slightly modified. In most cases, the reduce function emits a single value. Furthermore, implementations of MapReduce such as Hadoop [6] allow multiple MapReduce steps to run in succession. That is, the output of one MapReduce step is the input to a next MapReduce step. To reflect successive MapReduce steps in the programming model, each output value of the reduce function is associated with a key. Thus, the reduce function outputs key-value pairs; and as said, typically just one. For these reasons, we can rewrite the definition of the reduce function as follows:

```
reduce : (k_2, list(v_2)) → (k_3, v_3)
```

In Chapter 4 we transform this definition into an incremental counterpart, which incrementally applies the output key-value pair to an already aggregate value and, thus, supports real-time analytics.

In addition to the programming model, MapReduce consists of an execution framework. The execution framework executes map and reduce functions on a number of nodes. There is a master node and several worker nodes. The master node assigns tasks to worker nodes and overlooks the computation to recognize node failures. Worker nodes execute tasks in parallel.

The execution of MapReduce happens in two phases: a map phase and a reduce phase. Figure 2.3 illustrates the execution. In the map phase, the input data is

---

3Figure 2.3 closely resembles Figure 1 in the original MapReduce paper [32].

11
split into $M$ partitions ($M = 5$ in Figure 2.3). These partitions are processed by $M$ map tasks on a number of worker nodes. A map task invokes the map function for each key-value pair in the partition, and splits the output into $R$ intermediate partitions ($R = 2$ in Figure 2.3). Intermediate partitions are stored on the local file system of the worker node. After completion of all map tasks, the map phase is finished. In the reduce phase, these $R$ partitions are processed by $R$ reduce tasks on a number of worker nodes. A reduce task reads its assigned intermediate partitions, invokes the reduce function, and writes the output to the global file system. The combined output of all reduce tasks forms the final result.

**High Scalability.** MapReduce scales well with the number of nodes because the computation is distributed across the nodes. The more nodes are added, the more map and reduce tasks can run in parallel. In practice, MapReduce is able to analyze petabytes of data on thousands of nodes [33, 109].

**Fault Tolerance.** The execution framework of MapReduce tolerates failures of worker nodes. Whenever a worker node fails, all map tasks that were scheduled to run on this node are rescheduled to run on other nodes. The results of these map tasks are considered to be lost because they were stored on the node’s local file system. However, these results will be reproduced by the rescheduled map tasks. Similarly, all unfinished reduce tasks are rescheduled to run on other nodes. The results of unfinished tasks are considered to be lost, but will be reproduced by the rescheduled reduce tasks. The results of finished reduce tasks are not lost because they were written to the global file system, which is fault-tolerant itself. Thus, all

\footnote{The parameters $M$ and $R$ are specified by the user.}
tasks will eventually execute successfully, and results are always available in the global filesystem.

The master node presents a single point of failure. However, by replicating the state of the master node across several nodes the master node can be made fault-tolerant as well. State replication is for example used in distributed lock services such as Chubby [23].

The main disadvantage of MapReduce, with respect to real-time analytics, is poor data freshness. This is because MapReduce is batch-oriented and cannot update data incrementally. MapReduce analyzes data in batches: large chunks of data that are processed at once by map and reduce tasks. This leads to poor data freshness because results are only visible after the whole batch has been processed. Research projects, such as the Hadoop Online Prototype [30], try to increase data freshness by approximating results after parts of the batch have been processed; however, these early results are only an approximation and not one hundred percent correct. Furthermore, and more importantly, MapReduce cannot reuse already existing results and apply new data incrementally. On arrival of new data, both, the existing data and the new data have to be processed to produce up-to-date results. This, again, leads to poor data freshness. We will confirm this point experimentally in Chapter 4. Both of these points, processing data in batches and no support for incremental updates, lead to poor data freshness. Thus, MapReduce as such is not suited for data analytics in real time.

In summary, MapReduce [32] has gained significant importance in the area of data analytics because of three successful properties: the simple programming model, high scalability, and fault tolerance. However, MapReduce processes data in batches and is not suited for analyzing data in real time. The fact that MapReduce organizes data as key-value pairs lets us combine MapReduce with key-value stores to build a system for real-time analytics, which updates data incrementally and preserves all successful properties of MapReduce.

### 2.3 XML and Semi-structured Data

The second part of this thesis presents a new approach to data integration using XML and XML-related technologies.

XML (or written out Extensible Markup Language) is a flexible format for interchanging data. The XML standard [19] defines how to serialize data in a format understood by different devices and systems. The focus of XML is usability over the Internet; that is, messages formatted in XML can be sent over the Internet and will be understood by any system that implements the XML standard. Thereby,
XML is flexible enough to serialize a wide variety of data, ranging from office documents such as Microsoft Word documents to arbitrary object structures used in web services. This flexibility comes at a price however. Data from the same domain, for example personal data in businesses or patient records in the health care industry, can be serialized as XML in many different ways—or, in other words, in completely different schemas.

In XML, a document is logically composed of declarations, elements, comments, character references, and processing instructions [19]. All of these are indicated in the document by explicit markup [19]. Declarations, in the beginning of the document, define the version of the XML and the character encoding used, and contain or point to a grammar that defines the schema of the document (a DTD, see below). All these declarations are optional; however, an XML document should begin with a declaration that specifies the version of XML being used [19]. Elements are the main building block of XML documents. Elements have a name and can contain both atomic content such as text and numbers, and complex content such as other (nested) elements, comments, and processing instructions. Elements may also be empty. Each XML document contains exactly one document element, called the root. The name of an element may be qualified with a namespace to distinguish similar names from different data sources. (Qualified element names are prepended with a namespace prefix that must be associated with a namespace URI [17].) Furthermore, elements may be attached with attributes, which have a name and contain atomic content only. Processing instructions, character references, and comments are specific types of XML constructs, which will not be further discussed. A description of these can be found in the XML standard [19].

Figure 2.4 shows two examples of XML documents. Both documents contain a declaration that defines that XML version 1.0 is used. The document in Fig-
Figure 2.4(a) consists of a document element with name p:person, and two nested elements with names p:name and p:age. The name of an element is between two angle brackets—a markup construct called tag (e.g., <p:person> in Figure 2.4(a)). The content of an element is everything between the start tag (<p:person>) and the end tag (<p:person>). The namespace prefix p of the names of all elements is associated with the namespace URI http://people.com (see line 3 of the XML document in Figure 2.4(a)). An attribute is always inside an element’s start tag (e.g., pid="17" in Figure 2.4). Figure 2.4(b) consists of elements with names hl:patient-record, hl:patient-name, and hl:medical-history. These qualified names are associated with the namespace URI http://hl7.org.

A schema defines the structure and content of an XML document. In detail, a schema defines all of the following: (1) the names and namespaces of elements and attributes, (2) the types of their content (e.g., strings, numbers, complex content), and (3) in which order elements are nested. Furthermore, schemas may define constraints on elements and their content (e.g., foreign-key relationships or the maximum number of children of an element). There exist many standards to define the schema of an XML document; for example, XML Schema [42] or Document Type Definition (DTD) [18]. Figure 2.5 uses a simple pseudo notation to show the schemas of the XML documents in Figure 2.4. Figure 2.5 illustrates which names and types of elements and attributes are legal, and in which order elements are nested. (The at sign in @pid denotes that this is an attribute.) The schemas in Figure 2.5 do not define any constraints.

XML documents containing same or similar data may have completely different schemas. For example, both documents in Figure 2.4 contain personal information about the same person, Anna Lund. However, the schemas of these documents are different: they have different namespace URIs (http://people.com and http://hl7.org) and different element names for the name of the person (name and patient-name). Processing (XML) documents with different schemas is difficult, especially when the number of different schemas increases. In Chapter 6, we show that our new approach, Mapping Data to Queries, can quickly process documents in many different schemas.

Important for this thesis is that every XML document can be represented as a
labeled tree. In detail, every part of an XML document can be viewed as a node in a labeled tree; which means there are element nodes, attribute nodes, and text nodes representing atomic content of elements. (Again, we omit processing instructions and comments.) An element node has a label, which represents the name of the element. An attribute node has a label and a value, which represent the name and the value of the attribute. A text node has only a value, which represents the atomic content of an element. Furthermore, each tree representation has an empty root node. Nesting of elements and the belonging of attributes and atomic content to elements is represented as directed edges in the tree. In other words, a directed edge represents a child-of relationship. Many systems that process XML data use such a labeled tree representation of XML internally, for example XQuery processors such as Saxon [103].

Figure 2.6(a) illustrates how the XML document of Figure 2.4(a) is represented as a labeled tree. All elements of the XML document (p:person, p:name, and p:age) are nodes in the tree. Similarly, the attribute @pid is a node in the tree with value “17”. The two texts “Anna Lund” and “28” are also represented as nodes. Additionally, there is an empty root node. The directed edges represent child-of relationships between nodes. For example, p:name is a child of p:person.

Figure 2.6(b) shows the same tree as in Figure 2.6(a), except for one small but essential difference. In Figure 2.6(b), the names of elements are not assigned to the nodes of the tree, but to the incoming edges of these nodes. That is, the name of an incoming edge defines the name of the represented element. This difference is only small, but it allows to play some data integration tricks as we will show in Chapter 6. (Essentially, it is possible to introduce additional edges with different labels between existing nodes.)
In summary, XML is a flexible data format that allows to serialize many different types of documents and object structures. This flexibility, however, leads to XML documents that contain the same or similar information in different schemas. This is where data integration approaches are needed. One observation with respect to our new approach to data integration is that every XML document can be represented as a labeled tree, in which the labels of incoming edges define the names of the represented elements.

Our new approach to data integration, Mapping Data to Queries, is not bound to XML specifically, but can be adopted to any data format that can be represented as a labeled tree. Such data formats are typically called semi-structured data formats, with XML being one prominent representative. Other semi-structured data formats include JSON [31] and YAML [9]. All formats have in common that they explicitly describe data using markup (i.e., tags or other markers), and enforce hierarchies by nesting data elements.

2.4 XQuery

XQuery [12] is a programming language to query and modify XML data. XQuery was specifically built to natively handle XML data. That is, XQuery allows to navigate through XML structures, to filter elements by content, and to create new elements. The main concepts of XQuery are XPath path expressions, the FLOWR expression, and expressions for element construction. In detail, XPath path expressions [29] allow to address specific entities of an XML document. The FLOWR expression, named after its five clauses for, let, where, order by, and return, extends path expressions by means of joins, variable bindings, and ordering possibilities. The FLOWR expression can be seen as the XQuery equivalent to SQL's select-from-where clause. Last, XQuery allows to create new XML elements using element constructors. XQuery became popular as a query language for XML, much like SQL is the query language for relational data. However, extensions to XQuery have been developed for full-text search, in-place updates, and procedural programming (i.e., scripting). In this thesis, we focus on the querying capabilities of XQuery.

A simple example of a query written in XQuery is the following:

```xml
declare namespace p="http://people.com";
for $p in //p:person
where $p/p:age > 21
return <result>{$p}</result>
```
This query returns all persons aged above 21. If applied to the document in Figure 2.4(a), it would return the p:person element of Anna Lund, wrapped in a result element. In detail, the query consists of a prolog (line 1) and a query body (lines 2–4). The prolog consists of a namespace declaration, which associates the namespace prefix p with the namespace URI http://people.com. The query body consists of a FLOWR expression, which contains two XPath expressions and an element constructor. The XPath expression //p:person selects all p:person elements in an XML document, while the XPath expression $p/p:age selects the p:age child element of the element bound by variable $p. The enclosing FLOWR expression returns the p:person element (bound by $p) if the (numerical) content of p:age is larger than 21. Last, an element constructor creates a new result element that wraps every p:person element that is output.

Important for this thesis are the path expressions of XQuery for navigation within XML documents. XQuery, and more precisely XPath, has path expressions for navigating in an XML tree. These path expressions—also called location steps—consist of an axis, which specifies the relationship between nodes; and a node test, which specifies the name of elements to be selected. Typically, locations steps are notated in abbreviated syntax. For example, the location step ./p:age is actually an abbreviation for self::node()/child::p:age and selects all element children of the context node with name p:age. Similarly, the location step ./p:person is short for self::node()/descendant-or-self::node()/child::p:person and selects all p:person element descendants of the context node; that is, the children and the children of the children and so on. Furthermore, a location step may also have a predicate, which uses expressions to refine the set of selected nodes. For example, the location step //p:person/p:address[1] has the predicate [1], which selects the first address of all the addresses of a person.

In this thesis, the path expressions for navigation are important because we will interweave the evaluation of these path expressions with our new mechanism for data integration. That is, during the enumeration of nodes that are selected by a location step, we will integrate XML data with different schemas—thus, integrating data at the runtime of a query.

### 2.5 YFilter

The last important XML-related technology we build on is YFilter [38]. YFilter is an XML filtering system that provides fast matching of XML documents to

---

5The expression //p:person (without the leading period) selects any p:person element in the document, even the document node if it matches the node test p:person.
large numbers of XPath expressions. It allows to quickly determine if an XML document matches any XPath expression. In other words, YFilter is an index for XPath expressions. YFilter uses a subset of XPath 1.0 expressions with two axes, the child and descendant-or-self axes, and predicates only.

YFilter encodes a set of XPath expressions as a non-deterministic finite automaton (NFA). An XML document matches an XPath expression if the NFA reaches an accepting state. In detail, it works as follows: The nodes of an XML tree are input into the NFA in a top-down approach; that is, in depth-first left-to-right order. The NFA transitions to new states depending on the names of the nodes. Whenever an accepting state of the NFA is reached, the XML document matches the corresponding XPath expression. YFilter outputs success.

Figure 2.7 shows an example of a non-deterministic finite automaton built by YFilter. The NFA in Figure 2.7 corresponds to the single XPath expression //p:person/p:age. The NFA consists of an $\epsilon$-transition from states $q_0$ to $q_1$ and a self-loop from $q_1$ to $q_1$ to model the descendant-or-self axis //.. The transition between states $q_1$ and $q_2$ has the label p:person to only choose p:person elements. The transition between states $q_2$ and $q_3$ has the label p:age to choose a p:age child of p:person. State $q_3$ is an accepting state. When inserting the nodes of the XML document on the left of Figure 2.4 into this NFA, it would eventually reach the accepting state $q_3$ and output that this XML document matches the XPath expression //p:person/p:age.

The advantage of YFilter is that multiple XPath expressions can be indexed as a single NFA. With this technique, YFilter can quickly match XML documents to thousands of XPath expressions [38].

We use the YFilter technique in this thesis to quickly determine if a mapping rule matches an XML document that is currently processed. Mapping rules establish connections between different schemas. If there are thousands of schemas, there are several thousands of mapping rules. To match thousands of mapping rules to XML documents, YFilter will be greatly useful.

**Related Systems.** As an alternative to YFilter, there exist a variety of XML filtering systems based on different techniques and ideas. These XML filtering sys-
tems include XFilter [5] and BUFF [85], which are also based on NFAs; XPush [48], which is based on a deterministic pushdown automaton; AFilter [25], which is based on a number of custom-built data structures; and FiST [73], which is based on Prüfer sequences.

NFA-based XML filtering systems encode XPath expressions as non-deterministic automatons (as explained above). The first such system was XFilter [5]. XFilter is the predecessor of YFilter and encodes each XPath expression as a separate NFA. Thus, multiple XPath expressions result in multiple, separate NFAs, which all need to be evaluated when matching an XML document. In XFilter, XML documents are input into each NFA top-down (depth-first left-to-right), starting from the root. XFilter shows good performance for a small number of XPath expressions, but does not scale well with the number of XPath expressions [5]. YFilter extends XFilter in that it exploits common prefixes of XPath expressions and maintains only a single NFA. Consequently, YFilter is faster than XFilter when matching large numbers of XPath expressions against an XML document. YFilter scales better with the number of XPath expressions than XFilter [38].

In contrast to the top-down approach of XFilter and YFilter, BUFF [85] employs a bottom-up approach. That is, BUFF generates an NFA of all XPath expressions in reverse order and inputs nodes into the NFA starting from the leaf nodes of the XML document (using a backtracking algorithm). BUFF, therefore, exploits common suffixes of XPath expressions. BUFF is only slightly faster than YFilter, up to a factor of 1.3 [85].

In contrast to the previous NFA-based systems, XPush [48] uses a single deterministic pushdown automaton to filter XML documents. A pushdown automaton extends a finite automaton with a stack: transitions between states are based on the input as well as the top of the stack. A pushdown automaton is also able to modify the stack as part of a transition. XPush uses lookup tables to determine, based on the input and the top of the stack, which state to transition to. In XPush, lookup tables are computed lazily to improve performance. Unfortunately, the performance of XPush has not been compared to YFilter [48].

AFilter [25] uses custom-built data structures to filter XML documents: a number of stacks are filled with stack objects that reference each other. Furthermore, graphs and tries are used to encode XPath expressions. A stack object corresponds to one observed start tag of an XML document that is read. Yet, a start tag of an XML document may cause multiple stack objects to be put on the stack. Stack objects reference each other, via pointers, to maintain parent-child relationships of the XML. The advantage of AFilter is that it exploits common prefixes and suffixes, and enumerates states lazily. Furthermore, it allows to trade in memory for performance by disabling certain optimizations and throwing away the data.
structures necessary for these optimizations. AFilter outperforms YFilter by a factor of 3 to 5 depending on the workload [25].

FiST [73] uses a yet another approach to filtering XML documents. FiST encodes both, XML documents and XPath expressions, as Prüfer sequences. Prüfer sequences provide a one-to-one correspondence between labeled trees and a sequence of labels. If a labeled tree is a subtree of another labeled tree, then the Prüfer sequence of the former tree is a subsequence of the Prüfer sequence of the latter tree. FiST extends Prüfer sequences for the encoding of XML documents and XPath expressions. FiST filters XML documents by first comparing Prüfer sequences of XML documents with the Prüfer sequences of all XPath expressions and then running a post-processing step to filter out false positives. FiST outperforms YFilter by a factor of up to 2, and scales better than YFilter as the sizes of XML documents increase [73].

All of these approaches could be employed as part of Mapping Data to Queries in Chapter 6. In this work, we chose YFilter for two practical reasons. First, because of YFilter’s top-down approach, it is easy to interweave the evaluation of path expressions within Saxon [103] (the XQuery processor which we extend with MDQ) with YFilter. YFilter inputs nodes of an XML tree in depth-first left-to-right order in exactly the same way as descendant path expressions (and all other path expressions that recursively traverse the labeled tree representation of XML) are evaluated in Saxon. Other alternatives, especially BUFF and AFilter, would require a significant amount of implementation effort to interweave their XML filtering algorithms with the evaluation of path expressions. Second, YFilter proved to provide good performance. As we will show in Chapter 6, Mapping Data to Queries, which employs YFilter in one of its components, provides good performance (similar and better than traditional approaches), and scales well with the number of schemas of XML documents. We acknowledge that by using faster XML filtering techniques, such as AFilter, the performance of MDQ could be improved. (However, YFilter is only a part of MDQ and, thus, using a better XML filtering technique will improve the performance of MDQ only slightly.)
Chapter 3

Building Data Flows Using Key-Value Stores

In this chapter, we study how to build propagational data flows using key-value stores. Propagational data flows are especially present in social networks, which propagate updates from users to other users in real time. We build on key-value stores because they are typically used by social networks to store data in a scalable and available manner. The results of this chapter form the basis of our real-time analytics system of the next chapter.

The contents of this chapter have been published as a technical report [57].

3.1 Introduction

Social communication features of most of today’s largest websites require propagating the data inside the database. Incoming data is stored redundantly such that read requests can be answered efficiently. Such data propagation leads to a large number of propagational data flows.

More precisely, incoming data is a sequence of input tuples \((tb, k, v)\) where \(tb\) is a database table, \(k\) is a key, and \(v\) is the value associated with the key. Each input tuple is written to the database. Writing a tuple \((tb, k, v)\) means storing the key-value pair \((k, v)\) in table \(tb\) of the database. Each key is unique, such that key-value pairs never overwrite each other—key-value pairs are always appended to a database table. Multiple writes of the same tuple \((tb, k, v)\) will have the same effect as just one write of tuple \((tb, k, v)\). In other words, subsequent writes of a tuple that has been written before are idempotent.
A propagational data flow is defined as a mapping of one input tuple to a set $T$ of new tuples where each new tuple in $T$ is written to the database. The set $T$ of new tuples may contain any number of tuples. Each new tuple may have different values for $tb$, $k$, and $v$. In particular, new tuples can have different values for $tb$ and, thus, the key-value pair $(k, v)$ is propagated to different tables in the database. A new tuple must not overwrite existing tuples. Again, multiple writes of a tuple have the same effect as just one write.

The size of a propagational data flow is defined as the size of the set $T$ of new tuples, which are written to the database.

In this chapter, the input tuples and tuples of propagational data flows are written to the database with the following consistency properties:

- The write of an input tuple is read-your-write-consistent. That is, the process that has written the input tuple always reads the up-to-date value of the tuple, and never sees an older value [113].

- Each write of a tuple of a propagational data flow is eventual-consistent. That is, if no new updates are made to the tuple, eventually all accesses will return the last updated value [113].

In other words, after an input tuple has been written, it is immediately visible to the user who inserted the tuple. Tuples written as part of a propagational data flow may not be immediately visible to any user.

Propagational data flows are, for example, present in Twitter. On Twitter, users share small status updates (called tweets) with their friends (called followers). Each tweet initiates a propagational data flow that propagates the tweet to the user’s followers, inserts it into Twitter lists, and indexes it for full text search [8]. The propagation of tweets (i.e., the redundant storage of tweets) allows fast read accesses by users. Tweets are input tuples, which must be immediately visible to the user who posted the tweet (read-your-write consistency). However, the propagation can be delayed; not all followers see new tweets of their friends immediately (eventual consistency). All tweets of Twitter’s millions of users have to be propagated through the system, creating millions of propagational data flows. As different users have different numbers of followers, the sizes of propagational data flows vary.

Building a system that supports propagational data flows is not trivial. There are many requirements to the system that must be fulfilled:

1. The system must sustain a high throughput of input tuples per second.
2. Each write of an input tuple must be answered within a (guaranteed) low response time.⁶

3. The system must be robust to handle short periods of bursts of input tuples per second.

4. The system must be fault-tolerant. No tuples must be lost.

5. The system must scale as the number of input tuples per second increases.

In particular, the second requirement states that each user request must be answered within a low and guaranteed response time, even if the sizes of propagational data flows vary. That is, the size of the propagational data flow should not have an impact on response time. For example on Twitter, the number of followers of users varies a lot. Celebrities typically have many more followers than normal people. Depending on the user, different amounts of tuples need to be propagated. Still, the system must guarantee low response time for every write of an input tuple. The third requirement states that the system must be robust to handle short periods of bursts. For example, during Barack Obama’s inauguration Twitter experienced five times the normal load of tweets per second for several minutes [105]. The last requirement states that the system must be able to scale as the throughput of input tuples per second increases, and so can accommodate a growing user base. On Twitter, the number of active users is still growing (as explained in the introduction of this thesis).

On the other hand, the mentioned properties that subsequent writes of tuples are idempotent and that writes of propagational data flows are eventual consistent simplify the system implementation:

1. The system does not need to implement exactly-once semantics. As subsequent writes of tuples are idempotent, it suffices to implement at-least-once semantics.

2. The system does not need to write multiple tuples atomically. Because all writes of propagational data flows need only be eventual consistent, the system does not need to bundle the input tuple and corresponding tuples of the propagational data flow as one atomic transaction. Hence, transactions are not needed.

These points allow us to build propagational data flows on top of modern key-value stores, such as Cassandra [74], that typically do not provide exactly-once semantics.⁷

⁶Response time is defined as the time until a write of an input tuple is acknowledged to the user.
semantics and transactions. In this chapter, we assume that the key-value store has a notion of tables, in which key-value pairs can be stored.

There are several approaches to build propagational data flows (all of which guarantee at-least-once semantics). The first, naive approach writes all tuples (i.e., the input tuple and all tuples of the propagational data flow) in one batch, synchronously. As we will show, this approach sustains high throughput and provides good scalability but fails to guarantee response time and cannot handle bursts. The second approach is based on an external queue. The input tuple is written to the key-value store and buffered in the external queue. The corresponding tuples of the propagational data flow are written after the write of the input tuple has been acknowledged—asynchronously. Among others, this approach is used in Twitter [96]. As we will show, it overcomes the drawbacks of the naive approach in that it guarantees low response time and handles bursts. However, it needs increasingly more nodes (for application and queue) in order to scale as the number of input tuples per second increases. We propose a third approach that tightly integrates the queue into the key-value store: the queue is partitioned across the nodes of the key-value store. Tuples of the propagational data flow are also written asynchronously, after the write of input tuple has been acknowledged. This integrated approach provides better resource utilization compared to the external queue approach, because it reduces network traffic and better utilizes the CPUs of the nodes of the key-value store. Traditionally, products combine database and queue components for consistency reasons and easier application development [94, 115], but do not tightly integrate these components for better resource utilization.

In this chapter, we make the following contributions. We study the above listed approaches to build propagational data flows using key-value stores. We focus on the programming model, execution model, failure model, and scalability in the number of input tuples per second. We compare performance characteristics of all approaches varying essential workload parameters. We conclude that the integrated approach provides a number of advantages, the most significant being better resource utilization and therefore reduced cost of scalability in comparison to the external queue approach.

The remainder of this chapter is organized as follows. Section 3.2 presents an example of a propagational data flow, which we use throughout this chapter. Sections 3.3, 3.4, and 3.5 discuss the synchronous, external queue, and integrated approaches respectively. Section 3.6 compares these approaches experimentally. Section 3.7 discusses related work. Section 3.8 concludes this chapter.
3.2 Running Example

To discuss the alternative approaches to build propagational data flows, we use a simple propagational data flow as running example. We will discuss a more sophisticated propagational data flow in our experimental study. In this propagational data flow, illustrated in Figure 3.1, a user posts an input tuple $t_A = ("A", 17, "hello")$ to the application, where "A" is the table name, 17 is the unique identifier of the tuple, and "hello" is the message the user sends. The application requires $t_A$ to be written before the user request is allowed to return—the write of $t_A$ must be read-your-write-consistent. Furthermore, input tuple $t_A$ maps to two new tuples $t_B = ("B", 17, "hello")$ and $t_C = ("C", 17, "hello")$. These tuples are written to tables $B$ and $C$ respectively. The writes of tuples $t_B$ and $t_C$ are eventual-consistent.

Throughout the remainder of this chapter, we will refer to the write of $t_A$ as critical write, because it needs to be read-your-write-consistent; to the writes of $t_B$ and $t_C$, we will refer to as non-critical writes, because they only need to be eventual-consistent.

3.3 Synchronous Approach

The first, naive approach to implementing a propagational data flow is to write of all tuples, the input tuple and all tuples of the propagational data flow, in a single batch, synchronously. This is the standard setup of a web application that uses, for

![Diagram](image-url)
example, an Apache web server on top a key-value store. The application will write all tuples to the key-value store and only after the last tuple has been written the initial user request will return. Figure 3.2(a) displays the synchronous approach to the execution of the propagational data flow of Figure 3.1. The application uses multiple application threads to handle separate user requests. One thread handles one user request and writes all tuples of the respective propagational data flow to the key-value store sequentially. In Figure 3.2(a), one thread writes $t_A$, $t_B$, and $t_C$ to the key-value store. After the last write has finished the application thread acknowledges the initial user request.

A slightly different strategy of the synchronous approach is to write the tuples of a propagational data flow in parallel using multiple threads. However, this strategy suffers from the same disadvantage as the studied strategy where all tuples of a propagational data flow are written by just one thread which is high and fluctuating response times of user requests. This is because a propagational
data flow consisting of many tuples would use all available threads while other user requests are blocked and will be only acknowledged after all previous tuples have been written.

The advantage of the synchronous approach is that no modifications or extensions to the application or key-value store are needed. The disadvantage of this approach is the high response time of the user request. Despite the rather small critical write \( t_A \), the user has to wait for a long time until all other non-critical writes (\( t_B \) and \( t_C \)) have been completed. This is particularly harmful for social networking applications in which a user request typically causes many successive writes to the key-value store to propagate the data to all the user’s connections in the social network.

3.3.1 Programming and Execution Models

The synchronous approach requires no special programming model. All updates are executed sequentially from a standard client application; for example, a Java Servlet that connects to the key-value store using a standard API.

The execution model is based on the standard web application architecture with no additional components and no changes to the key-value store.

3.3.2 Failure Model

In case of failures the developer of the application needs to take care that there are no inconsistencies. Inconsistencies occur, for example, when there is a failure after tuples \( t_A \) and \( t_B \) have been successfully written but before tuple \( t_C \) has been written. The failure will be visible to the application because the request to write \( t_C \) will return an error, or not return at all and the application will experience a time out. In this case there are two options to fix it: (1) retry to write the missing tuples \( (t_C \text{ in this example}) \) or (2) delete all tuples that had been written so far \( (t_A \text{ and } t_B) \).

3.3.3 Scalability

This approach scales with the number of input tuples per second using standard techniques supported by modern application servers and key-value stores, such as load balancing and data partitioning.
3.4 External Queue Approach

The high response time of the synchronous approach can be reduced by only acknowledging the critical write to the user. All non-critical writes are executed after the acknowledgement, asynchronously. The state-of-the-art approach to implementing asynchronous execution is to use an external queue and workers. In addition to the write to the key-value store, the user’s input tuple is buffered in the queue before sending the acknowledgement to the user. The queues are drained by workers. Workers implement the propagational data flow: they map the input tuple to a set of new tuples and write these new tuples to the key-value store. The advantage is that the user request can return much faster compared to the synchronous approach. The disadvantage is that the propagated tuples become visible some time after the original user request; however, this is acceptable as discussed in the introduction. Other disadvantages of this approach are higher costs for setup and maintenance as it requires an external queue and workers, which typically reside on separate machines, and a more complicated programming model and failure handling as we discuss below.

Figure 3.2(b) displays the differences to the previous approach. An external queue and workers are added to the system. The application writes tuple $t_A$ to the key-value store and, potentially in parallel, pushes $t_A$ to the queue. After these two steps the application is able to acknowledge the user request. One worker thread (denoted by $W$ in Figure 3.2(b)) reads tuple $t_A$ from the queue and writes tuples $t_B$ and $t_C$ to the key-value store.

3.4.1 Programming Model

Using an external queue, the application developer must obey the following rules. First, the application has to push an input tuple to the queue after successfully writing it to the key-value store. Second, the application programmer has to implement workers that read tuples from the queue and implement the propagational data flow (i.e., map the input tuple to new tuples and write these to the key-value store). To optimize response time, the input tuple can be pushed to the queue in parallel to the write to the key-value store. We used this optimization in our experiments in Section 3.6.

3.4.2 Execution Model

There need to be additional machines for the external queue and workers, and make them interact as described above. For small workloads, workers can run on
the same machine as the application. But for real-world applications and larger workloads, they have to run on separate machines to avoid overloading the application server (as we will show in our experiments). As in the synchronous approach, the execution model does not require any changes to the key-value store.

3.4.3 Failure Model

The developer has to fight failures at two fronts, within the application and within workers. We assume that the application writes the critical tuple $t_A$ to the key-value store and to the queue in parallel. Therefore, if the write to the key-value store results in an error or the write is not acknowledged, tuple $t_A$ may have already been pushed to the queue. Vice versa, if the write to the queue results in an error or the write is not acknowledged, tuple $t_A$ may have already been written to the key-value store. Therefore, we can only retry writing to the key-value store or to the queue in case of failures. Otherwise it would lead to an inconsistent state of the data: $t_A$ has been written, but the corresponding tuples $t_B$ and $t_C$ will never exist—or $t_A$ has not been written but $t_B$ and $t_C$ exist. We can only report an error to the user in the case when both writes, to the key-value store and to the queue, fail.

Once the critical tuple $t_A$ has been pushed to the queue, we need to make sure that the non-critical tuples $t_B$ and $t_C$ will be written to the key-value store under any circumstances. Within workers, two different failure cases need to be considered. The first case is if the worker quits working after it has read the tuple from the queue and before it was able to complete the job. In this case, another worker cannot repeat the work of the failed worker because tuple $t_A$ has been deleted from the queue; tuples $t_B$ and $t_C$ are lost. The second case is if the worker is not able to write to the key-value store because of either an error response from the key-value store or no response at all.

To deal with the first failure of lost tuples, queue implementations typically allow to read tuple tentatively. A worker thread reads a tuple tentatively and if the read is not confirmed by the worker at a later point in time, the tuple will stay in the queue and be visible again after a specific time-out interval. Thus, another worker is able to repeat the job. In between the tentative read and the confirmation, the tuple cannot be read by any other workers to prevent multiple workers from doing the same work. If a worker crashes after it has written some or all non-critical data tuples to the key-value store but before the confirmation, another worker will read the non-confirmed tuple from the queue and repeat the writes of the non-critical tuples. As discussed, writing tuples multiple times presents no problem because subsequent writes of the same tuple are idempotent.
To deal with the second failure, in which the worker does not crash but the writes to the key-value store respond with an error or with no response at all, the worker needs to retry the writes until successful. The worker cannot inform the application of such an error, because the originating user request has already been acknowledged. Therefore, it remains to the worker to retry writing to the key-value store until successful (or log the error).

### 3.4.4 Scalability

This approach, as well, scales with the number of input tuples per second. By adding more nodes to the system, the throughput of input tuples per second the system is able to sustain increases. However, scaling this approach means to increase the number of nodes not only for the application and key-value store but also for the queue and workers. As a result, it requires increasingly more resources, in particular network and CPU, to scale. This drawback will be eliminated by our integrated approach described in the next section.

### 3.5 Integrated Approach

In this section, we describe an optimization of the external queue approach that aims at better resource utilization and, furthermore, provides minor advantages in development and maintenance of the system. There are two basic design decisions behind this approach:

1. The queue is distributed across the nodes of the key-value store. Each node maintains its own queues and worker threads.

2. Fault tolerance (guaranteeing at-least-once semantics) is implemented by reusing the replication mechanism of the key-value store.

In detail, each node maintains queues that buffer tasks. A task is defined as a tuple \((tb, k, v, f)\) where \(tb\) is a database table, \(k\) is a key, \(v\) is a value, and \(f\) is a pointer to a function. A task is buffered in the queue of the node that is contacted to write an input tuple. Worker threads read tasks from the queues and call the function \(f\). Function \(f\) implements the propagational data flow and is parameterized with tuple \((tb, k, v)\). Function \(f\) maps tuple \((tb, k, v)\) to a set of new tuples and writes them to the key-value store. Function \(f\) has no return value. Tasks are executed asynchronously, some time after the write of the input tuple has been acknowledged.
To guarantee at-least-once semantics, we reuse the replication mechanism of the key-value store. Replica nodes buffer unfinished tasks in and execute them only in case the original node that was scheduled to execute these tasks fails. We explain the necessary protocols in detail below.

Figure 3.2(c) illustrates the integrated approach. In contrast to the external queue approach, there are no queue and workers outside of the key-value store. In Figure 3.2(c), the application writes tuple $t_A$ to the key-value store. The propagational data flow, which writes $t_B$ and $t_C$, is executed entirely inside the key-value store via the execution of tasks.

Figure 3.3 shows the execution model of the integrated approach within the key-value store. In Figure 3.3(a), input tuple $t_A$ is written to node $N_0$. Node $N_0$ is the contacted node. It forwards tuple $t_A$ to the replication group (which consists of nodes $N_1$ to $N_3$) and buffers a task $tsk$ in one of its queues. In Figure 3.3(b), the task is executed and writes $t_B$ and $t_C$ to their respective replication groups (which consist of nodes $N_2$ to $N_4$ in this example).

The basic mechanism of this approach is task execution in response to a write to the key-value store. Such a mechanism can be classified as triggers. Triggers are executed asynchronously in our integrated approach. In relational database systems, triggers consist of four parts: a name, a definition when the triggers is executed, a limiting condition, and an implementation [67]. In our approach, a
trigger only has two parts: a table name and an implementation. A trigger is always executed after a write to the specified table. The implementation consists of the function $f$, which implements the propagational data flow. Triggers must be implemented as idempotent operations: repeated trigger execution must have the same effect as a single execution.

The main advantage of the integrated approach is better resource utilization in comparison to the external queue approach while having the same performance characteristics. Distributing queues and workers across the nodes of the key-value store removes the network and CPU cost of communicating with the external queue. As we will show in the experiments in Section 3.6, the integrated approach utilizes the network and CPU resources of the nodes of the system more efficiently and handles the same throughput (i.e., input tuples per second) as the external queue approach with fewer nodes.

Furthermore, the integrated approach has an easier programming model and better maintenance of the system than the external queue approach. The programming model is easier because the application developer only writes the critical input tuple to the key-value store and defines a trigger, but does not need to ensure fault tolerance himself. The system is easier to maintain than the one of the external queue approach because the system of the external queue approach contains additional machines for external queue and workers, which need additional maintenance.

The disadvantage of this approach is that requires changes to the nodes of the key-value store. Each node needs to be extended with queues and worker threads, and execute the master/slave protocol explained below.

Because this approach tightly integrates queues into the key-value store, we first specify assumptions about the key-value store that are required to implement the integrated approach.

- All tables of the key-value store are partitioned and replicated across nodes by key. We call the component of the key-value store that replicates data replication subsystem and the component that stores data storage subsystem.

- The key-value store has an interface to read a value $v$ for a given table $tb$ and key $k$.

- An application may contact any node of the key-value store to read a value or write an input tuple. We call the contacted node the master.\footnote{Typically, in key-value stores, the contacted node is called update coordinator. Because we extend the key-value store with additional protocols, we use a different terminology.}

Because this approach tightly integrates queues into the key-value store, we first specify assumptions about the key-value store that are required to implement the integrated approach.
• The key-value store does not support transactions.

• Each node of the key-value store is notified about failures of any other node. We call the component that notifies nodes of failures *gossiping subsystem*.

• Each node of the key-value store knows all triggers.

Note that these assumptions are quite general and exist in practically every modern key-value store; for example, in Cassandra [74] used in our experiments.

### 3.5.1 Programming Model

The programming model of the integrated approach consists of two parts: (1) writing the input tuple to the key-value store, and (2) specifying the propagational data flow as a set of triggers. The application writes the input tuple to the key-value store. If it does not get a response from the key-value store, it retries writing the input tuple at a different node of the key-value store. Specifying the propagational data flow as a set of triggers is easy because all aspects of asynchronous execution and fault tolerance are transparent to the application developer.

To define a trigger, the application developer needs to specify the table, on which the trigger registered on, and provide the implementation (the function $f$) of the trigger. As said, the function $f$ is parameterized with the database table $tb$, the key $k$, and the value $v$ of an input tuple. Given these parameters the trigger implementation has all the knowledge of the write that caused the execution of the trigger. Of course, the implementation of a trigger may also read from the key-value store. As said, triggers must be implemented as idempotent operations.

### 3.5.2 Execution Model

This approach requires changes to the nodes of the key-value store. Each node of the key-value store is extended with all of the following:

1. Handlers that execute the master/slave protocol defined below.

2. Queues that buffer tasks. There is one queue per trigger.

3. Worker threads that execute tasks.

4. A backup task map that contains \((node, task)\) pairs of uncompleted tasks.
Figure 3.4 illustrates the architecture of one node of the integrated approach. On top of the original node of the key-value store, we add a master handler, a slave handler, a queue for each trigger, worker threads for each queue, and a backup task map. The master handler executes the master part the master/slave protocol (described below) and communicates with the replication and storage subsystems of the key-value store and inserts tasks into queues. Queues are drained by worker threads (denoted by \( T \) in Figure 3.4), which communicate with the replication subsystem. Writes of tuples from within a worker thread are sent to the replication subsystem and internally forwarded to the storage subsystem. The slave handler executes the slave part of the master/slave protocol and communicates with the replication and storage subsystems and stores \((node, task)\) pairs in the backup task map. A \((node, task)\) pair associates an uncompleted task with the node on which the task was scheduled to run. The gossiping subsystem, which is part of the original key-value store architecture, notifies the slave handler of node failures in the key-value store.

The input to the node can be either a write request of an input tuple or a read request. A write request arrives at the master handler and is answered with an acknowledgement (ACK). A read request arrives at the original key-value store components and is answered as defined by the underlying key-value store. The read request consists of a table \( tb \) and key \( k \), and as a result the value for key \( k \) in table \( tb \) is returned. For completeness, Figure 3.4 shows that the replication and gossiping subsystems communicate with the replication and gossiping subsystems of other nodes respectively.

The master/slave protocol defines in which order messages are sent and tasks are executed. The protocol ensures at-least-once semantics of the execution of tasks. That is, tasks are guaranteed to be executed at least once, even in the presence of node failures. The terms master and slave are defined as follows:

Master: The master is the node of the key-value store that is contacted by the application to write an input tuple.

Slave: A slave is a node of the key-value store that is part of the replication group that stores the input tuple and is not the master.

The application may contact different nodes of the system to write in input tuple. Thus, different nodes may act as master for different input tuples. The master may be part of the replication group that stores the input tuple. In that case, it is only the master and not a slave.

The number of slaves depends on the size of the replication group and is typically small. Usually, the size of a replication group is set to 3 in practical scenarios [34].
Figure 3.4: Architecture of one node of the integrated approach.

The master/slave protocol is given in figures 3.5 and 3.6.

**Protocol of the Master.** Tasks are executed at the master for any write request of an input tuple. For each write request of an input tuple, the master forwards the input tuple to the slaves and finds all triggers that are registered on the input tuple’s table. For each trigger, it inserts a task into the queue that corresponds to the trigger. The tasks consists of the values of the input tuple $t_b, k, v$, and the function $f$ of the respective trigger. The master executes tasks and send notification messages of successful completion of task execution to the slaves. The protocol of the master is shown in Figure 3.5.

In Step $M_2$ of the master protocol, the master waits for acknowledgments from the slaves. There is synchronicity here to ensure that the input tuple has been successfully written to the slaves. Yet, tasks are executed asynchronously some time after the original input tuple has been written.

**Protocol of a Slave.** Slaves store the input tuple locally (because they are part of the replication group) and maintain a backup task map, which contains $(node, task)$ pairs that associate uncompleted tasks with the node on which the task was scheduled to be executed. If a slave receives a notification from the master
On reception of a write request of an input tuple \((tb, k, v)\)

\[ M_1. \] Send \((tb, k, v)\) to slaves and store \((tb, k, v)\) locally if the master is part of the replication group.

\[ M_2. \] Wait for acknowledgements from slaves. After a timeout, send an error to the application and return.

\[ M_3. \] Find triggers for table \(tb \rightarrow T\).

\[ M_4. \] For each trigger \(tr \in T\): submit a task \((tb, k, v, f_{tr})\) into the corresponding queue of \(tr\).

\[ M_5. \] Send acknowledgement to the application.

On execution of a task \((tb, k, v, f_{tr})\)

\[ M_1^* \] Execute function \(f_{tr}\) with parameters \(tb, k, v\).

\[ M_2^* \] Send notification message of successful completion of task \((tb, k, v, f_{tr})\) to slaves.

\[ M_3^* \] Wait for acknowledgements of notification message from slaves. Repeat step \(M_2^*\) after a timeout.

Figure 3.5: Protocol of the master.

that the task has been completed, it removes the respective \((node, task)\) pair from the backup task map. If a slave is notified of a node failure by the gossiping subsystem, it executes all tasks in the backup task map that were scheduled to be executed on the failed node. In other words, when a node fails, all slaves execute the backup tasks for which the failed node was the master. (Because tasks are executed multiple times, triggers need to be idempotent.) The protocol of the slave is shown in Figure 3.6.

3.5.3 Failure Model

The presented protocols ensure at-least-once semantics of task execution. That is, all tasks will be executed at least once, even in the presence of node failures. There are two types of node failures: failure of the master and failure of a slave. In the following, we will discuss that at-least-once semantics are guaranteed independent of the time when a node failure occurs.

**Failure of the Master.** Given the protocol in Figure 3.5, guaranteeing at-least-once semantics consists of two steps: first, making sure that the task is written to
On reception of a message to replicate tuple \((tb, k, v)\) from master node \(n\)

\[ S_1. \] Store \((tb, k, v)\) locally.

\[ S_2. \] Find triggers for table \(tb \rightarrow T\).

\[ S_3. \] For each trigger \(tr \in T\): put a pair \((n, (tb, k, v, f_{tr}))\) into the backup task map.

\[ S_4. \] Send acknowledgement to the master.

On reception of a notification message of completion of task \((tb, k, v, f_{tr})\)
from master node \(n\)

\[ S'_1. \] Remove pair \((n, (tb, k, v, f_{tr}))\) from backup task map.

\[ S'_2. \] Send acknowledgement to the master.

On notification of a failure of node \(n\)

\[ S'_1. \] Find all pairs associated with node \(n\) in the backup task map \(\rightarrow P\).

\[ S'_2. \] For each pair \((n, (tb, k, v, f)) \in P\): Execute function \(f\) with parameters \(tb, k, v\).

\[ S'_3. \] For each pair \((n, (tb, k, v, f)) \in P\): Remove \((n, (tb, k, v, f))\) from backup task map.

Figure 3.6: Protocol of a slave.

the queue, and second, making sure that the task is executed.

On reception of a write request, if the master fails before sending the acknowledgment to the application (anywhere before step \(M_5\) in Figure 3.5), the task might or might not have written to the queue. Because the application will not get a response, the application retries the write request (at a different node) until it gets an acknowledgement. Therefore, at some point, we will reach step \(M_5\) in the master protocol and are sure to have scheduled the task in the queue.

If the master fails after having sent the acknowledgement to the application (after step \(M_5\)) the task is stored in the queue but might not have been executed. The application will not retry the write request because the request has been acknowledged. Because the master goes down after step \(M_5\), we know that the slaves have put the task into their backup task maps (because the master received their acknowledgements). The slaves will be notified by the gossiping subsystem that the master has failed (see the list of assumptions about the key-value store). In that case, each slave will execute all tasks in the backup task map that correspond
to the master node. Thus, the task will be executed at least once.

In particular, for any task the following holds. If the master fails before executing the task (before step $M_1^*$), the task is executed by the slaves that acknowledged step $M_1$. If the master fails after the execution but before sending the notifications to the slaves (in between steps $M_1^*$ and $M_2^*$) the task is executed by the master and all slaves. If the master fails in the middle of sending notification messages (in the middle of $M_2^*$) the task is executed by the master and those slaves that did not receive a notification message from the master.

Our master/slave protocol allows fail-stop semantics of node failures. When the master recovers from a failure, it will start with a clean state. All queues of the master will be empty. All tasks that were stored in these queues when the master failed are lost. This is acceptable because the slaves executed those tasks.

**Failure of a Slave.** If a slave fails during any of the steps of the protocol in Figure 3.6, it will not harm the at-least-once semantics of execution of tasks. If the slave fails before step $S_4$, the master will time out in step $M_2$ and the application will retry writing the input tuple until it gets an acknowledgment (and, as discussed, the task will be executed). If the slave fails after step $S_4$, the master will execute the task itself.

Because of fail-stop semantics of node failures, when the slave recovers from a failure its backup task map will be empty. All $(node, task)$ pairs stored in this map are lost. It is possible that a restarted slave receives a notification message without a corresponding $(node, task)$ pair stored in the map. In that case it will acknowledge the message without deleting any pair.

The system tolerates up to $s - 1$ failures of slaves including the master where $s$ is the number of slaves. If $s$ nodes fails (including the master) some tasks will never be executed.

### 3.5.4 Scalability

The integrated approach scales with the number of input tuples per second. The more nodes are added to the system, the higher the throughput of input tuples per second that can be processed. As we will show, this approach requires less nodes than the external queue approach to handle the same amount of throughput.

### 3.6 Experiments and Results

In order to evaluate the different approaches, we modeled the real-world workload of Twitter. For that we use the detailed analyses of Twitter presented by Krish-
namurthy et al. [70] and Kwak et al. [72]. In Twitter, whenever a user posts a
tweet, it is propagated to all followers of this user. The recent study by Kwak et
al. [72] states that the number of followers per user follows a highly skewed power
law distribution: most users have few followers but some users have hundreds of
thousands of followers. We used the distribution from [72] to generate the mapping
of users to their followers.

In our experiments, the system is divided into three parts: application, queue,
and database (key-value store). The application acts as load generator and insert
tweets into the key-value store at a given rate. The key-value store contains three
tables: tweets, followers, and timeline. The tweets table stores all tweets a user has
published. The followers table contains the mapping from users to followers. The
timeline table contains a timeline for each user. The timeline of a user consists of
all of his tweets, and those of all his followers. On Twitter, a user usually reads
the content of his own timeline.

The propagational data flow in our experiments is implemented by a trigger that is
set of the tweets table. The trigger implementation is the following: for each new
tweet, the trigger first reads the followers table to find all followers of the author
of the tweet and then appends the tweet to the timeline of each of the followers
of the author. In other words, each incoming tweet is propagated to all follower
timelines (according to the mapping in the followers table).

We implemented all approaches in Java 1.6 on top of Cassandra [74], a popular
key-value store used in Twitter, Facebook, Digg, and many other web applications.
Cassandra implements a shared-nothing architecture with data partitioned and
replicated across nodes. All assumptions required to build the integrated approach
listed in Section 3.5 hold for Cassandra. In particular, Cassandra implements a
gossip protocol to notify all nodes about a failure of another node.

For the synchronous approach and the external queue approach, Cassandra was
used without modifications.

For the integrated approach, we implemented the master/slave protocol described
in Section 3.5. We modified Cassandra’s update coordinator code to execute the
master protocol each time an input tuple is written to the tweets table in Cassan-
dra. To optimize step $M_2^*$ of the master/slave protocol, we modified the backup
task map of a slave to maintain a relation $md5((tb,k,v,f)) \rightarrow (n,(tb,k,v,f))$.
The function $md5$ computes the MD5 hash of a task $(tb,k,v,f)$. In step $M_2^*$, we
only send the MD5 hash of task $(tb,k,v,f)$, which results in a smaller notification
message.\footnote{There is a small $\frac{n^2}{n^m}$ probability ($n$ being the number of entries of the backup task map and
typically small) that a hash will collide with an existing hash in the backup task map, which
results in overwriting tasks. Overwritten tasks would never be executed if corresponding master}
integrated approach to 1000 for stable experimental results. Otherwise it would be possible to write (almost) indefinitely many tasks to the queue and would lead to unstable results. In the experiment with bursts, we increased the size of queues to 100’000.

In all experiments, the replication factor of Cassandra was set to 2. The write consistency level was set to \textit{ALL}, the read consistency level to \textit{ONE}. That is, all replicas must be successfully updated to acknowledge a write, but only one node is needs to be contacted to read from. As external queue, we used Kestrel [68]. Kestrel is the queue implementation used in Twitter [96].

All experiments were carried out using machines with an AMD Opteron 2.4 GHz CPU and 6 GB RAM running Ubuntu Server 10. Our experimental setup consisted of four machines: one machine runs the application generating the tweets to be inserted; one machine runs Kestrel, the queue needed in the external queue approach; and two machines run Cassandra, the key-value store. In the scalability experiment, we increase the number of machines up to a total number of 21 machines. In all experiments, we ran the benchmark for 30 minutes. We report on throughput in messages per second and the median, standard deviation, and maximum of response time in milliseconds. One message per second means that the application was able to insert one tweet into the key-value store, query all followers, and propagate the tweet to these followers. The size of each tweet in our experiments is about 200 bytes. Response time measures the time until the application acknowledges the tweet insertion. For the burst and fault-tolerance experiments, we sampled throughput and response time every three seconds and show data collected over a period of three minutes. Furthermore, we collected performance metrics and report on the average CPU load and the sum of incoming and outgoing network traffic (including traffic within the key-value store) of each part of the system.

3.6.1 Saturating Workload

In the first experiment we measure the maximum throughput each approach is able to sustain. The application issues as many tweets as possible to saturate the key-value store. Table 3.1 shows throughput and response times of the saturating workload. We make the following observations: (1) All approaches peak at the same maximum throughput of about 1300 messages per second. The limiting factor is the write capacity of the key-value store. (2) Communicating with the external queue does not decrease the maximum throughput of the external queue approach. (3) Integrating queues into the key-value store does not decrease the node failed. However, due to the small probability, we consider this optimization practical.
Table 3.1: Performance of the studied approaches. (2 nodes)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Throughput [msg/s]</th>
<th>Response Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>stddev</td>
</tr>
<tr>
<td><strong>Saturating workload</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>1297</td>
<td>20.66</td>
</tr>
<tr>
<td>External Queue</td>
<td>1291</td>
<td>0.83</td>
</tr>
<tr>
<td>Integrated</td>
<td>1289</td>
<td>1.26</td>
</tr>
<tr>
<td><strong>80% workload</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>1000</td>
<td>5.35</td>
</tr>
<tr>
<td>External Queue</td>
<td>1000</td>
<td>0.84</td>
</tr>
<tr>
<td>Integrated</td>
<td>1000</td>
<td>0.80</td>
</tr>
</tbody>
</table>

maximum throughput that can be achieved. The additional processing overhead that is shifted from the application to the key-value store is negligible for Twitter’s data propagation workload. (4) When the key-value store is saturated, response time cannot be measured accurately as discussed in standard textbooks [64]. Thus, we will study the response time of a non-saturating workload.

### 3.6.2 80% Workload

In this experiment we throttle the number of issued tweets to about 80% of the maximum throughput to not saturate the key-value store. The application issues 1000 tweets per second. The results are also shown in Table 3.1. We make the following observations: (1) All approaches handle 1000 messages per second. (2) The median response time of the synchronous approach is about 7 times higher than that of the other approaches. In the synchronous approach, the application will acknowledge a tweet only after it has propagated the tweet to all followers. Because of the highly skewed workload (i.e., few users have hundreds of thousands of followers), we observe a high standard deviation and a maximum response time of 13.8 seconds, which is unacceptable for a web application. (3) The median response time of the external queue approach and that of the integrated approach are the same. Both asynchronous approaches have a guaranteed (low) response time below 0.2 seconds.
3.6.3 Bursty Workload

In addition to guaranteed low response time, another advantage of asynchronous execution of propagational data flows is being able to handle short periods of bursts of requests. Bursts may even exceed the maximum capacity of the key-value store because subsequent operations can be buffered using queues. In this experiment, the rate at which the application issues tweets is the following. First, the application issues tweets at a rate of about 50% of the saturating capacity (600 messages per second). After 60 seconds the request rate jumps to 200% of the saturating capacity (2600 messages per second). This burst lasts for 30 seconds and afterwards the request rate drops back to 50%.

Figure 3.7(a) shows the number of messages each approach handles over time. We observe that asynchronous approaches can handle short bursts of requests. User tweets can be written at a rate of 2600 messages per second while subsequent writes to propagate the tweets to the followers are buffered. The tweets will be propagated as soon as there is enough capacity (not shown in Figure 3.7(a)). The synchronous approach is only able to handle requests at the rate of the maximum capacity of 1300 message per second only. It has to drop requests in case of bursts. The asynchronous approaches did not drop messages in this experiment. Figure 3.7(b) shows the measured response times. For the asynchronous approaches the response time remains constant. For the synchronous approach, the response time jumps from about 6ms to 25ms because the system is saturated during the burst of requests.

We conclude that (1) asynchronous approaches are able to handle bursts without dropping messages, and (2) the integrated approach shows the same throughput and response time characteristics as the external queue approach.

3.6.4 Fault Tolerance

To observe the fault-tolerance behavior of all approaches, in this experiment we run a key-value store consisting of 4 nodes. The application issues a load of 1000 messages per second. 60 seconds into the experiment, we stop 1 node by manually killing the key-value store’s process on this machine. 30 seconds later, we restart the process. Figure 3.8 shows the throughput and response time for all approaches. For about 5 seconds after the node has faulted, all nodes of the key-value store do not respond. The throughput drops to 0 and requests are stalled until the system responds again. Afterwards, the throughput fluctuates as the key-value store tries to recover from the fault. Throughput stabilizes at around 350 messages per second 20 seconds after the fault. Once we restart the faulted process, it takes the key-value store another 20 to 30 seconds to get back to a stable handling of the issued
Figure 3.7: Tolerance to a 30 second burst. Throughput (a) and response time (b). (2 nodes)

load. The response times of the asynchronous approaches fluctuate less than that of the synchronous approach. In all cases we manually checked that all inserted tweets were propagated to the respective followers.

### 3.6.5 Scalability

In the last experiment we study the scalability of the different approaches and measure resource utilization. We vary the number of nodes of the key-value store and measure the maximum throughput in messages per second each configuration is able to sustain. We adjust the number of nodes of the application and queue as necessary to obtain best results for each configuration.

Figure 3.9(a) shows the maximum throughput of each approach when varying the number of nodes. All approaches scale equally well (i.e., sustain the same through-
Figure 3.8: Tolerance to a 30 second node failure. Throughput (a) and response time (b). (4 nodes)

put) when increasing the number of nodes. Figure 3.9(b) shows the response times of each approach for an 80% workload and varying number of nodes. The load was set to 80% of the throughput in Figure 3.9(a). We observe that the response times remain constant for each approach as the size of the key-value store increases.

Table 3.2 shows the resource utilization for 2 nodes and 16 nodes when the nodes of the key-value store are saturated (Figure 3.9(a)). Here, we make the following observations: (1) The application of the synchronous approach has high CPU load and network traffic because it is responsible for sending all tuples across the network to the key-value store. (2) CPU load and network traffic of the application of the external queue approach is even higher than that of the synchronous approach because of the communication with the queue. (3) CPU and network traffic of the integrated approach is lowest compared to the other approaches because the application only sends the critical input tuple to the key-value store. The rest of
the propagational data flow is executed within the key-value store, which saves communication overhead and has no noticeable CPU overhead. Main memory usage was not a limiting factor in our experiments.\textsuperscript{9} The number of nodes to perform the experiments is lowest for the integrated approach. For 16 nodes of the key-value store, the synchronous approach needed 3 application nodes to completely saturate the key-value store. The external queue approach needed 3 application nodes (1 load generator and 2 queue workers) and also 2 queue nodes. The integrated approach needed only 2 application nodes. To conclude, when scaling the key-value store to 16 nodes the integrated approach saves 26.2 MB/s in network traffic compared to the external queue approach, which is a reduction of 35%; and requires only a total of 18 nodes instead of 21 nodes, which is a reduction of almost 15%.

\textsuperscript{9}The measurements of main memory utilization can be found in our technical report [57].

Figure 3.9: Scalability with the number of nodes. Maximum throughput when saturating the nodes (a), response time of an 80\% workload (b). (2 nodes)
Table 3.2: Resource utilization for 2 and 16 nodes. (Saturating workload)

<table>
<thead>
<tr>
<th>Approach</th>
<th>CPU [%]</th>
<th>Network [MB/s]</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>App DB Queue</td>
<td>App DB Queue</td>
<td>App DB Queue Sum</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>11.0</td>
<td>39.6</td>
<td>2.1</td>
</tr>
<tr>
<td>Ext. Queue</td>
<td>21.6</td>
<td>36.4</td>
<td>16.6</td>
</tr>
<tr>
<td>Integrated</td>
<td>2.3</td>
<td>40.9</td>
<td>–</td>
</tr>
<tr>
<td>16 Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>22.1</td>
<td>41.9</td>
<td>–</td>
</tr>
<tr>
<td>Ext. Queue</td>
<td>38.8</td>
<td>41.9</td>
<td>33.2</td>
</tr>
<tr>
<td>Integrated</td>
<td>6.2</td>
<td>41.1</td>
<td>–</td>
</tr>
</tbody>
</table>

3.7 Related Work

Information dissemination systems (also called publish/subscribe systems), such as SIFT [118], LeSubscribe [39], SASE [116], and Cayuga [35], distribute documents to users based on user profiles (or subscriptions). For example, a user may submit a user profile “fly ∧ fishing” (∧ being the boolean and operator), and any document that contains these words will be forwarded to the user [118]. Research on information dissemination systems primarily focuses on efficiently matching an incoming document to large numbers of user profiles. In particular, SIFT and LeSubscribe build indexes of user profiles to efficiently match individual documents. SASE and Cayuga extend these systems with more powerful subscription languages that span multiple documents, aggregate documents, and so on [116, 35]. XML filtering systems, such as [5, 38, 85, 48, 25, 73] discussed in Section 2.5, can also be classified as information dissemination systems. Here, user profiles are defined as XPath expressions, and documents that are matched are XML documents. All of this work studies how to efficiently match documents against user profiles; the forwarding (or propagation) of matching documents is not studied. However, all of this work could be leveraged to build an information dissemination system that utilizes resources efficiently: implementing one of the approaches (e.g., SIFT) as a trigger of our integrated approach would allow a fast matching of user profiles and efficient utilization of resources.\(^{10}\)

Microsoft SQL Server Service Broker [115] and Oracle Advanced Queuing [94] combine database and queues into a single system. Both products highlight that

---

\(^{10}\)Assuming matched documents are propagated within the database and are not sent out, for example, via email.
it results lower maintenance and administration costs because of the removal of expensive message-oriented middleware products. Service Broker also highlights easier programmability because of a unified programming model, which allows reading and writing to queues using extensions to the Transact-SQL language of Microsoft SQL Server. Furthermore, Service Broker reports on better performance characteristics, but does not provide performance numbers [115]. In Service Broker and Oracle Advanced Queuing, read and write operations to the queue are part of database transactions. Therefore, both products guarantee exactly-once delivery and retrieval of messages from queues. Both products allow to implement complex business process workflows (more complex than the studied propagational data flow in this chapter), however, both products do not elaborate to which extend database and queues are combined (i.e., if it more resembles the external queue approach or more resembles the integrated approach of this chapter). Furthermore, they do not report on performance numbers and in particular do not address resource utilization.

The Demaq system [13, 14] allows to build distributed applications that are based on the asynchronous exchange of (XML) messages. Demaq proposes a fully declarative rule language to specify an application as a set of rules and a runtime system that automatically optimizes the execution of the application. Application rules are executed asynchronously after a write of a message to a queue, and each message is processed exactly once. The advantages of Demaq over a standard application server with an imperative programming model are easier programmability because enqueuing and dequeuing are part of the language semantics, automatic handling of message persistency and recovery, better performance because of automatic optimization and because there is no multi-layer (e.g., three-tier) environment as in traditional application servers. Demaq is built on top of the Natix native XML database system [43], which natively supports queue-based storage for XML messages [14].

Percolator [97] is a system for real-time analytics recently developed by Google. Percolator is built on top of Bigtable [27] and uses observers and tasks to process data in real time. Similar to triggers in our integrated approach, observers react to changes in the database. Whenever a user-specified column in the database changes, observers trigger the execution of tasks. However, Percolator only guarantees at-most-once semantics of task execution—not all changes to the user-specified column result in the execution of a task. Thus, implementing propagational data flows using Percolator would result in some tuples not being propagated correctly. In contrast, our integrated approach guarantees at-least-once semantics of task execution—all tuples will be propagated correctly. We will study Google Percolator in more detail in the next chapter.
3.8 Conclusion

In this chapter, we studied various approaches to build propagational data flows. We showed that asynchronous mechanisms are key when building propagational data flows as it guarantees low response time and allows to handle short periods of bursts of input tuples per second. Our experiments confirm that the state-of-the-art approach based on an external queue provides desirable performance characteristics but does not utilize resources efficiently.

We addressed this issue by integrating queues into the key-value store. The propagational data flow happens inside the key-value store via the execution of tasks. This allows to scale the system with fewer nodes—utilizing resources more efficiently. We also showed that this integrated approach has the following minor advantages: (1) an easier programming model, which encapsulates all the details of asynchronous execution and failure handling, and (2) a tight integration of queues into the key-value store, which eliminates maintenance and administration cost of an external queue.

The key mechanisms of the integrated approach, asynchronous task execution and protocols for fault tolerance, have been proposed as an extension to the next version of the Cassandra key-value store.\footnote{Cassandra Ticket 1311, \url{https://issues.apache.org/jira/browse/cassandra-1311}}
Chapter 4

The Limmat System

Based on the previous findings, in this chapter we present Limmat, a system for real-time analytics. The previous chapter showed that integrating queues into the key-value store utilizes resources more efficiently. In this chapter, we extend this approach with an easier, MapReduce-like programming model, and transactions and synchronization for strong consistency. Thereby we study important design space options that must be considered when building a real-time analytics system.

We demonstrated Limmat at the 2011 VLDB conference [46].

4.1 Introduction

Since its introduction in 2004, MapReduce [32] has gained significant importance in the area of data analytics. The reasons of its success are the simple programming model, the ability to tolerate machine failures, and high scalability. In detail, MapReduce offers a simple, yet powerful programming model with which it is possible to specify a wide variety of analytical work flows [33]. The fault-tolerance property of MapReduce allows to continue data processing in the light of machine failures, and results are always available. This in turn enables a cost-effective installation of MapReduce on commodity hardware. Finally, MapReduce scales tremendously well, allowing to analyze petabytes of data using thousands of machines [33, 109].

A downside of MapReduce is that it is batch-oriented; it cannot update results incrementally. On arrival of new data, all of the data—the existing data and the new data—have to be processed to produce up-to-date results. This leads to poor data freshness. To prove this point, we ran a simple experiment that compares the data freshness of Hadoop [6], a batch-oriented MapReduce implementation,
to Limmat, the real-time analytics system we present in this chapter. Limmat updates data incrementally. In this experiment we measure the time it takes to produce up-to-date results when 2MB of new data are appended to existing data in the gigabyte range. (The complete setup of this experiment is described in Section 4.7.) Upon arrival of new data, Hadoop analyzes both existing and new data to produce up-to-date results. Limmat only analyzes the new data and incrementally updates the existing results of the previous analytics. Figure 4.1 illustrates the results of this experiment. There are two important findings: (1) Hadoop needs much more time than Limmat to produce up-to-date results; and (2) for Hadoop, the update time depends on the size of the existing data, whereas for Limmat it does not. This shows that updating data incrementally is crucial to achieving good data freshness.

Bringing together the successful properties of MapReduce and the insight that incremental updates enable good data freshness, we study in this chapter how to build a real-time analytics system. In this chapter, a real-time analytics system must meet the following requirements:

1. Process data in real time
2. A simple programming model, close to MapReduce
3. Tolerate machine failures
4. Scale with the number of nodes
5. Strong consistency (as defined below)
The use cases of a real-time analytics system are the same as for traditional MapReduce, but in a real-time fashion. In particular, in this work we focus on counting events in real time (such as clicks on a web application or tweets on Twitter), generating trending topics (by sorting events/keywords based on their counts), and indexing data in real time. A complete application on top of Limmat, which includes counting and trending topics, is presented in the next chapter, Chapter 5.

One challenge in building a real-time analytics system is data consistency. In MapReduce, data consistency is (naturally) taken for granted. It is defined as: “a distributed implementation [of a MapReduce program] produces the same output as would have been produced by a non-faulting sequential execution of the entire program” [32]. Of course, a real-time analytics system should have the same characteristic. Similarly to the above definition, in this chapter we define strong consistency as:

**Strong consistency:** A distributed execution of a real-time analytics program produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

This definition has three important implications: exactly-once semantics, synchronization, and in-order processing. By having to produce the same output as a non-faulting execution, a distributed real-time analytics system must implement exactly-once semantics. Each map and reduce function of a real-time analytics program must be successfully executed exactly once for every data item. By having to produce the same output as a sequential execution, a distributed real-time analytics system must synchronize the execution of the reduce function by key (as in all existing MapReduce systems). Furthermore, a sequential execution of a MapReduce program defines an order in which data items are processed. The distributed real-time analytics system must guarantee that data items are processed in the same order.

However, in Limmat we do not guarantee in-order processing. There are three reasons. First, the use cases of counting and generating trending topics do not depend on the order in which events are input into the system. Second, guaranteeing in-order processing is considered “fundamentally unrealistic and unacceptable in most enterprise environments” [71]. And third, the closely related systems Twitter Storm [110] and Google Percolator [97] also do not guarantee in-order processing of data, which is an additional hint that in-order processing is less important for real-time analytics systems.

In summary, the contributions of this chapter are the following:

- We list the requirements and design space options to build a real-time analytics system, and classify related systems accordingly.
We discuss the design space options in detail and run experiments on the most important options to gain more insight into the trade-offs of the design space options.

As a result of this discussion, we present Limmat, a system for real-time analytics. Limmat allows to analyze data in real time, provides all successful properties of MapReduce, and is strongly consistent (defined above). The support of strong consistency sets Limmat apart from other real-time analytics systems, such as Google Percolator [97] and Twitter Storm [110].

The remainder of this chapter is structured as follows: Section 4.2 lists assumptions we make about the system and data. Section 4.3 introduces a simple, MapReduce-like programming model for incremental computation. Section 4.4 explores the design space to build a real-time analytics system. Section 4.5 reviews related work. Section 4.6 presents the design of Limmat. Section 4.7 presents the results of performance experiments. Section 4.8 concludes this chapter.

4.2 Assumptions

We make the following assumptions about the system and data:

- We assume a distributed system of many nodes.
- Nodes store data in the form of key-value pairs in tables.
- All tables of the system are partitioned and replicated across nodes by key.
- Nodes also process key-value pairs (in real time).

In detail, we assume a distributed system of many nodes, which communicate with each other using message passing. Nodes forward and store data in the form of key-value pairs. In particular, values can be randomly accessed by their key, avoiding global scans of data. Values do not necessarily have to be atomic, but can have complex types (e.g., serialized objects).

To store data, each node has a local storage component such as a hard disk. The local storage can be persistent (i.e., survive node failures and restarts) or non-persistent. We will explain the design implications of persistent vs. non-persistent storage in detail below.

As in the previous chapter, nodes process data via task execution upon changes to tables. The system does not guarantee in-order processing of data. That is,
it is not guaranteed that key-value pairs are processed in the order in which they arrive at the system. Truviso [71] argues that, in practice, data streams are rarely continuously ordered (and, thus, a system does not need to guarantee ordering of data inside the system) and that, still, systems correctly process and account for every input, no matter how late it arrives [71]. Therefore, we believe this assumption is reasonable and does not restrict the application space of our system too much.

4.3 Programming Model

In this work, we require a real-time analytics system to have a simple programming model, close to the original MapReduce programming model. Here, we present a modification of the MapReduce programming model, which enables incremental computation.

As discussed in Chapter 2, the original MapReduce programming model is batch-oriented. While the map function of MapReduce operates on single input values, the reduce function operates on batches. Reduce takes a list of input values (the batch), combines it into a single aggregate value, and outputs the aggregate value as result (see Figure 2.2 and the discussion in Section 2.2).

Inspired by Logothetis et al. [78], we modify this programming model to incrementally update results. In particular, we modify the reduce function. The modified function, \( \text{reduce}' \), incrementally applies the value of a new key-value pair to the already existing aggregate value (i.e., result) associated with the key of the pair. This modification allows to apply a new value to the corresponding aggregate value as soon as a new key-value pair arrives. We do not modify the map function as it already allows incremental processing. The difference between map and \( \text{reduce}' \) is that multiple map functions can be executed in parallel for the same key, while the execution of a \( \text{reduce}' \) function has to be synchronized for the same key to not modify the same aggregate value in parallel and, thus, to guarantee correctness of the computation.

Figure 4.2 depicts the modified programming model. The map function is the same as in the original MapReduce specification. The \( \text{reduce}' \) function takes as parameters a key, a value, and the existing aggregate value for that key. It outputs a key-value pair with the same key and the new aggregate value. The aggregate value does not necessarily need to be an atomic value (e.g., a counter), but can be a collection or any other type (e.g., a sorted list). The new aggregate value is output for each tuple (as with map). The \( \text{reduce}' \) function produces a stream of results, where a new aggregate value overwrites the older aggregate value. An application
map : \( (k_1, v_1) \rightarrow \text{list}(k_2, v_2) \)
reduce' : \( (k_2, v_2, \text{agg}_\text{old}) \rightarrow (k_2, \text{agg}_\text{new}) \)

Figure 4.2: MapReduce-like programming model that supports incremental updates of data.

outside of Limmat always reads the latest version of the aggregate value, for all key-value pairs that have been processed so far.

As an example of how the modified programming model works, consider again the word count application discussed in Section 2.2. The application counts the number of occurrences of each word in a collection of documents. Expressing this application in the modified programming model results in the following map/reduce' job:

input table: documents
output table: wordCounts

map(String key, String value):
    for each word w in value:
        emit (w, 1);

reduce'(String key, int value, int aggregate):
    int sum = aggregate + value;
    emit (key, sum);

A map/reduce' job consists of an input table, an output table, and implementations of a map and reduce' function. From the input table, data is input into the map function. To the output table, the output of the reduce' function is written. In this map/reduce' job, the implementation of the map function is the same as in Section 2.2 (and in [32]). The implementation of the reduce' function takes a count for a particular word and applies it to the existing aggregate value. It outputs a new aggregate value: the sum of the count and the aggregate value.

The flow of data in this model is the following: Whenever a new key-value pair is written to the input table, the corresponding map function is executed. The results of the map function are forwarded to the reduce' function, which is also supplied with the old aggregate value that is read from the output table for the key of the forwarded key-value pair. The key-value pair output by the reduce' (containing
the new aggregate value) is written to the output table. The new aggregate value will serve as input to the next call to the reduce function for the same key. In the map/reduce job above, whenever a key-value is written to the documents table, the map function is executed and, as a result, one or more reduce functions are executed. The output of the reduce function is written to the wordCounts table.

A table can serve as input to multiple map/reduce jobs. In that case, whenever a key-value pair is written to the table, multiple map jobs will be executed. The output table of each map/reduce job must be unique. There can be multiple map/reduce jobs in succession. In that case, the output table of one map/reduce job is defined as the input table of a next map/reduce job.

In summary, we modified the MapReduce programming model to be able to update data incrementally. The modifications were kept as small as possible to preserve the simplicity of the original programming model. The modified programming model allows to implement a wide variety of analytical computations in our experience.

4.4 Design Space

In this section, we explore the design space of a real-time analytics system. A real-time analytics system must satisfy the five requirements listed in the introduction. We use the same technique as in the previous chapter to fulfill Requirement 1 (processing data in real time): writes to tables trigger the execution of tasks, which execute map and reduce functions. A map task executes the map function of a map/reduce job for one key-value pair. Similarly, a reduce task executes the reduce function for one key-value pair and the corresponding aggregate value. Requirement 2 (a simple programming model) is satisfied by the modified map/reduce programming model of the previous section. Consequently, in the following, we study design space options to satisfy requirements 3 to 5: fault tolerance, scalability with the number of nodes, and strong consistency.

4.4.1 Fault Tolerance Requirement

By fault tolerance, we mean non-blocking execution and data availability in the event of node failures. The computation must not block when a node of the system fails, but continue as nothing had happened. Data must always be available, even if some nodes of the system are unavailable. Both of these points are interrelated: the computation can only continue if data is available; that is, if input data can be read and results can be written.
Given the programming model above, there are two types of data: (1) externally visible data (input data to map and result data produced by reduce'), and (2) intermediate data output by map and forwarded to reduce'. The result data is not necessarily the final result of the computation, but can be a snapshot of computation performed so far. The difference between these two types of data is that a user of the system is interested in externally visible data, whereas a user is typically not interested in intermediate data. For example, in the word count application from above, a user is interested in the original documents (the input data) and the word counts of words that have been counted so far (the result data), but the user is not interested in the key-value pairs output by map (the intermediate data). Therefore, externally visible data must always be available; intermediate data can be temporarily unavailable.

**Design Space Option 1: Replication of Externally Visible Data.** Replication of externally visible data is necessary to ensure availability. The data is replicated to multiple nodes to avoid that the data becomes unavailable when one node fails. Figure 4.3 shows the logical flow of the data. The node marked with i handles the insertion of new data and replicates it to multiple nodes. Similarly, the node marked with r, which executes the reduce' job, replicates the result data to multiple nodes. The nodes on which the data is replicated is called the replication group.

The strategy to replicate the data can either be quorum-based (as, e.g., in Amazon Dynamo [34]) or using full replication (as, e.g., in the Hadoop file system [6]). Both strategies have advantages and disadvantages. In quorum-based replication, read and write requests are sent to \( N \) nodes (\( N \) is the replication factor, i.e., the size of the replication group) and have to be acknowledged by at least \( R \) nodes for reads.
and $W$ nodes for writes. Setting $R + W > N$ ensures that always the most up-to-date version of the data is read. In Dynamo, a common $(N, R, W)$ configuration is $(3, 2, 2)$. In other words, reads and writes of data have to be acknowledged by at least the majority of nodes of the replication group. A $(3, 2, 2)$ configuration tolerates node failures as long as the majority of nodes is still available. There are complex protocols based on hinted handoffs to ensure data availability even if the majority of nodes is down [34]. In a $(3, 2, 2)$ configuration (and similar configurations with $R > 1$), read requests are expensive because multiple nodes have to be contacted to read the data. Full replication means writing to all nodes of the replication group, but reading only from one node. In that sense, it is a special case of quorum-based replication with $R = 1$ and $W = N$. Full replication allows faster reads of data because only one node needs to be contacted. But, if a node fails, writing data is not possible anymore. In Hadoop’s full replication strategy, if a node fails, writing data is only temporarily disabled until the data is re-replicated to other nodes that are still alive [6].

**Design Space Option 2: Replication of Intermediate Data.** Replication of intermediate data depends on the properties of the system; in particular, if storage is persistent or not. Replication of intermediate data is necessary if storage is non-persistent. Non-persistent storage implies fail-stop semantics: if a node fails and is restarted, all data is lost. In this case, intermediate data needs to be replicated for durability reasons, to survive the data loss when a node crashes. Replication is not necessary if storage is persistent. Persistent storage implies fail-recover semantics: if a node fails and is restarted, all data that was written before the failure still exists after the restart. Data survives but is temporarily unavailable.

In Figure 4.3, the node marked with $m$ executes a map job and forwards intermediate data to the node responsible to execute reduce$'$. Intermediate data may or may not be replicated, which is illustrated by the dotted arrows and system nodes. There are three important implications of not replicating intermediate data. (1) Intermediate data becomes temporarily unavailable during a node failure. This is acceptable because the user is not interested in intermediate data. (2) Intermediate data that is unavailable during a node failure will be processed only after the node is back up. This may lead to reordering of events, which we assume is acceptable. (3) Temporary unavailability leads to lower data freshness, which may or may not be acceptable depending on the application.

**4.4.2 Scalability Requirement**

Scalability with the number of nodes can be achieved by partitioning the execution of map and reduce$'$ functions across the nodes of the system. There are no
restrictions for the execution of map functions. Map functions can be executed in parallel on any node in the system, independent of key or value. In existing MapReduce systems, the execution of a reduce function is synchronized by key [32, 6]. That is, a reduce function is not allowed to be executed in parallel for values with the same key. Similarly, in this work, the execution of $\text{reduce}'$ needs to be synchronized by key. For each key, there must be at most one execution of the $\text{reduce}'$ function of a map/reduce$'$ job at any time. $\text{Reduce}'$ is not allowed to be executed in parallel for values with the same key. Otherwise, consistency of aggregate values is not guaranteed. For values with different keys, $\text{reduce}'$ can be executed in parallel. Therefore, the execution of $\text{reduce}'$ can also be partitioned across multiple nodes, for values with different keys. In summary, scalability is achieved by partitioning the computation by key.

Design Space Option 3: Distribution of Computation and Storage. A design space option that affects scalability is the distribution of computation and storage among nodes of the system. That is, on which nodes the computation is run and on which nodes data is stored. Essentially, there are two options. The straight-forward option is to divide the nodes of the system into nodes that execute map and reduce$'$ tasks and nodes that store data. The second option is to use every node for both, computation and storage. In this case, each node is responsible to store a partition of the data and also execute parts of the map and reduce$'$ tasks. Of course, there are other options to allocate computation and storage (e.g., hybrid approaches between the two options above depending on node properties), but we will not discuss these in detail here.

Figure 4.4 illustrates the two options. In Figure 4.4(a) there are separate nodes for computation and storage. The computation nodes will execute map and reduce$'$ tasks and communicate with the storage nodes to read and write data. The storage nodes communicate among each other to replicate data. In Figure 4.4(b) computation and storage are combined. Each node executes tasks and stores data. The nodes communicate with each other, forwarding data to other nodes and replicating data to maintain availability.

In both cases, there is a special node, the lock manager. The lock manager acts as the central coordinator for computation nodes to acquire and release locks. Locking is necessary to synchronize the execution of reduce$'$ tasks. The lock manager itself can be distributed for fault-tolerance reasons. (Communication with the lock manager is not shown in Figure 4.4.)

The advantage of separation of computation and storage is to make use of stand-alone products (e.g., databases and queues) that have proven to operate well under all conditions. Combining computation and storage almost always needs custom-built solutions.
The advantage of combining computation and storage is performance. It reduces the communication effort between nodes and allows to scale the system more efficiently. In the previous chapter, we showed that the CPUs of the storage nodes are under-utilized, allowing to perform computation on the nodes as well. We were able to show that the combining computation and storage reduces network traffic by 35% and requires 15% less nodes to scale with the with number of input tuples per second (see previous chapter), while maintaining all performance characteristics of the separated approach (i.e., throughput and response time characteristics, tolerance to failures and bursts). Based on these findings, we argue that the combined approach is better suited for a real-time analytics system.

Another advantage of the combined approach is local task synchronization without involving the lock manager too often. Combining storage and computation allows to assign each node a range of keys for which it is solely responsible; that is, for which it needs to store data (and replicate it to its neighbors), and execute map and reduce tasks. When executing reduce tasks, the node knows that it is the only node in the system that executes reduce for this key. Therefore the node can locally synchronize the execution of reduce and does not need to involve the lock manager for every execution. The lock manager is still needed to ensure non-
blocking execution in case of node failures. If the node fails, another node in the system needs to adopt the key range to continue to execute tasks. In this case, synchronization between nodes is required. This is achieved using fixed-length leases [45], which are acquired from the lock manager in regular (e.g., 30-second) intervals.

**Design Space Option 4: Support for Hotspots.** Another design space option that affects scalability with the number of nodes is the support for hotspots. Hotspots (or bottlenecks) are single points of the computation that are overloaded and cannot be trivially partitioned across nodes. Hotspots occur when too many values with the same key are processed by a reduce' function. So many values, in fact, that the (synchronized) execution of the reduce' function is slower than new values are arriving as input. Depending on the input data and application, the system may or may not need to support hotspots.

Hotspots can have two different causes. First, hotspots occur if the input data is skewed (i.e., not uniformly distributed). For example, when counting the number of occurrences of each word in a number of input documents and the documents mainly contain the word “foo”, the reduce' function that counts the word “foo” will be a hotspot. Second, hotspots occur if the specification of a map/reduce' program itself contains a hotspot. For example, a program that counts all words of all documents. Thus, there is only one counter for all words. The reduce' function that maintains this counter is a hotspot.

Implementing support for hotspots is not trivial. One strategy to support hotspots is to partition the hotspot; that is, to keep multiple aggregate values for the same key, and combine aggregate values at query time. To do this as transparently to the map/reduce' program as possible, each key of an intermediate key-value pair output by a map function is appended with a number drawn uniformly from 1 to $n$. ($n$ being the number of hotspot partitions.) Now there are $n$ reduce' tasks that can run in parallel (and possibly on different nodes) for originally the same key. At query time (i.e., when a user requests the aggregate value for a specific key), the $n$ internal aggregate values need to be combined into the final aggregate value. For this, there needs to be a user-defined function that combines aggregate values. That is, the programming model of Section 4.3 needs to be extended with a function that merges aggregate values:

$$merge : (agg_1, agg_2) \rightarrow agg_{final}$$

$merge$ may be called recursively to combine more than two aggregate values. This technique is transparent to map and reduce' implementations; these need not be modified. Only the new merge function needs to be implemented by the application developer.
In this strategy, however, *merge* itself may become a hotspot. That is the case when the final aggregate value that is computed by *merge* is the input to another map/reduce job. In that case, the final aggregate value needs to be computed whenever one of the aggregate value partitions changes. That is, *merge* is called many times and if *merge* is an expensive operation then it may become a hotspot itself. Of course, this depends on how expensive the *merge* operation is.

This strategy is similar to the proposal to implement distributed counters in the Cassandra key-value store [95]. There is also a Map-Reduce-Merge model [119] consisting of MapReduce plus an additional merge phase, which allow to express relational algebra operators and implement joins efficiently.

A much simpler strategy to deal with hotspots is to do load shedding; that is, to drop events when there is a hotspot. Depending on the application, this might lead to loss of consistency. In the example from above, when counting the number of occurrences of the word “foo”, the count of “foo” would be wrong. For applications that only need to see the latest event of a series of events (e.g., the latest version of a crawled website), load shedding is acceptable. Load shedding is for example implemented in Google Percolator [97].

Dealing with hotspots in a real-time analytics system is an interesting research problem. There are many challenges in supporting hotspots (e.g., making it as generic and transparent as possible, and maintaining low latency), and different solutions. For example, in interesting trade-off is to accumulate small, timely batches of input values and process data in batches. This introduces a latency/throughput trade-off that seems worth to be studied. However, an in-depth study of strategies to support hotspots is out of the scope of this thesis.

### 4.4.3 Strong Consistency Requirement

As outlined in the introduction, guaranteeing strong consistency means to implement exactly-once semantics and synchronization. Each map and reduce function must be successfully executed exactly once and, as discussed already, the execute of reduce function must be synchronized. In the following, we will discuss how to implement exactly-once semantics. (Implementing synchronization has been discussed in Design Space Option 3.)

**Design Space Option 5: Granularity of Transactions.** To implement exactly-once semantics, there needs to be support for atomicity. All operations of the execution of a map or reduce function has to happen atomically. In detail, the input of a map or reduce function needs to be acknowledged, the function has to be executed, and the output of it needs to be forwarded to the next task or replicated to multiple nodes; and all of that has to happen atomically (all or nothing).
The key to support atomicity is to use transactions. A transaction combines all operations of the execution of a map or reduce function and executes them as a single unit. In case of failures, partially executed transactions are rolled back. Either the transaction successfully commits and all operations were successfully carried out, or the transaction aborts and all operations are reset to their initial state as if nothing happened.

Transactions have two negative properties: high administrative cost and mutual blocking. A transaction has high administrative cost because it spans multiple nodes. (Outputs of map and reduce are forwarded to other nodes or replicated to multiple nodes, and therefore transactions span multiple nodes.) Transactions across multiple nodes are typically implemented using the two-phase commit protocol [100]. The two-phase commit protocol involves two rounds of messages between nodes, leading to a high administrative cost of a transaction. In addition, for data consistency, a transaction must hold a lock on each key of a reduce task it is executing. (Map tasks do not require locks because they can run in parallel for tasks with the same key.) Transactions that try to acquire locks for the same key block each other and can only run sequentially. This has a negative impact on parallel execution of transactions.

The design decision that must be made concerns the granularity of transactions. The two main options are (1) a large transaction that includes the execution of a map task and all subsequent reduce tasks, or (2) several smaller transactions, each of which includes a single map or reduce task. Figure 4.5 illustrates the different options. In Figure 4.5(a), one large transaction involves all tasks: the map task, which outputs three intermediate key-value pairs, and the three subsequent reduce tasks. In Figure 4.5(b), each task is wrapped in its own transaction. (Transaction boundaries are denoted by the dashed rectangle.) Mixes of these two options (e.g., transactions containing some, but not all, tasks) are possible, but will not be discussed in detail here.

There is a non-trivial trade-off between these two options of granularity: A few large transactions mean less administrative costs but a greater chance to collide with other transactions; several smaller transactions mean greater administrative costs but a smaller chance to collide. The administrative cost of a single transaction involving multiple tasks (Figure 4.5(a)) is less expensive because the two-phase commit protocol needs to be run only once. The cost is higher if there are several transactions (Figure 4.5(b)) because the two-phase commit protocol needs to be run several times involving more messages. On the other hand, a few large transactions are more likely to block each other because they each hold locks for several keys. Several smaller transactions, each with at most one lock on a key, are less likely to block each other; thus, more transactions can run in parallel. It is not clear which option has better performance characteristics. We will study this
Figure 4.5: Granularity of transactions. A single transaction for map and subsequent reduce′ tasks (a); or smaller transactions, one for each task (b).

trade-off in Section 4.7.

Furthermore, there are applications that do not require exactly-once semantics to produce (statistically) correct results; for example, web indexing and computing PageRank [20]. For these types of applications at-most-once semantics of task execution is sufficient because skipping some intermediate data items still leads to statistically correct results. Relaxing the requirement of exactly-once semantics raises the question, what is the overhead of exactly-once semantics compared to at-most-once semantics. We will also study this in Section 4.7.

4.4.4 Summary

The design space options discussed in this section result from the requirements to provide fault tolerance, scalability, and exactly-once semantics. Fault tolerance involves replication of data, scalability involves distribution of computation and support for hotspots, and exactly-once semantics involves transactions.

Table 4.1 summarizes the design space options discussed in this section: (1) Externally visible data (input and result data) must be replicated to guarantee data availability and non-blocking execution. There is no "no"-option. (2) Intermediate data (i.e., data that is forwarded from a map task to the next reduce′ task)
Table 4.1: Design space options.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication of externally visible data</td>
<td>Yes</td>
</tr>
<tr>
<td>Replication of intermediate data</td>
<td>Yes</td>
</tr>
<tr>
<td>Computation and storage</td>
<td>Separated</td>
</tr>
<tr>
<td>Support for hotspots</td>
<td>Yes</td>
</tr>
<tr>
<td>Granularity of transactions</td>
<td>Per task</td>
</tr>
</tbody>
</table>

does not necessarily need to be replicated, depending on the persistency of the storage. If storage is persistent, replication can be avoided. (3) Computation and storage may be separated or combined. Combining storage and computation allows to optimize synchronization and reduces network usage. (4) Depending on the data distribution and the computation, the system should support hotspots. Support for hotspots is not trivial and an interesting research avenue on its own. (5) Transactions, which are necessary to guarantee exactly-once semantics, may include only a single task or span multiple, connected tasks.

4.5 Related Work

We will now discuss recently developed real-time analytics systems with respect to the requirements of such systems and the design decisions. To repeat, the requirements of real-time analytics systems are (1) processing data in real time; (2) offering a simple, MapReduce-like programming model; (3) fault tolerance; (4) scalability with the number of nodes; and (5) strong consistency, in particular exactly-once semantics. The design space options were discussed in the previous section. Furthermore, we cover iterative MapReduce engines and data stream systems.

Percolator [97] is a real-time analytics system developed at Google. Percolator was built to prepare web pages for inclusion in the web search index, a task formerly carried out by a series of MapReduce jobs. Percolator updates data incrementally instead of processing data in batches. By indexing documents incrementally, the authors were able to drop the average age of a document that appears in the index by 50%. This highlights that updating data incrementally is crucial to achieving good data freshness; the main observation of the experiment shown in Figure 4.1. Percolator greatly motivates the need for incremental, real-time analytics systems.

In essence, Percolator consists of observers and tasks. Observers react to changes in the database. Whenever a (user-specified) database column changes, observers
trigger the execution of tasks. Tasks are implemented by the application developer, and are executed using ACID-compliant transactions—either all of the operations within a task are executed or none. However, Percolator does not guarantee that all changes to a column trigger an execution of a task.

With respect to the requirements of a real-time analytics system, Percolator has the following characteristics: (1) Percolator updates data in real time. It uses observers to trigger the execution of tasks whenever a database column changes. (2) The programming model is a normal program written in an imperative programming language. The application developer implements a task using C++. There is a Percolator API to start and commit transactions. This allows the developer a lot of freedom when implementing tasks. For example, transaction may span multiple keys/rows in the database. However, there is also some overhead. The developer must explicitly implement transactions. In Limmat, transactions (to synchronize the execution of tasks) are hidden from the programmer; the execution of reduce’ functions is automatically synchronized by Limmat’s execution framework. (3) Percolator provides fault tolerance by using transactions within a task and snapshot isolation. (4) The system scales by partitioning storage and computation by key. (5) Surprisingly, there is no exactly-once semantics for task execution. Instead, Percolator provides at-most-once semantics; some changes to a column may not trigger the execution of a task. However, this is reasonable for applications such as web indexing.

With respect to the design space options of Table 4.1, the authors of Percolator made the following decisions. Input and result data are replicated. Replication is done by storing the data in the Bigtable distributed database [27]. There is no intermediate data, which is written to the database. Intermediate data is kept within the state of a task. From reading the Percolator paper [97], it is not clear to which extend computation and storage are combined or separated. The Design section of the paper states that computation and storage processes run on the same machine and are tightly integrated; but the Experimental section suggests that the storage component, Bigtable, uses far more machines than the computation component. Synchronization is achieved using snapshot isolation, which involves an external timestamp service and a lock manager. Hotspots are dealt with by dropping events; not all changes trigger a task execution. Finally, transactions are on a per-task basis and do not span multiple tasks. But transactions are allowed to span multiple, different keys; and not only the key of the column that triggered the task execution.

Storm [110] is a real-time analytics system recently developed at Twitter. Storm was built with three use cases in mind: stream processing, continuous computations, and distributed remote procedure calls (RPC) [79]. Storm can process a stream of new data and update databases in real time. An example of continu-
ous computations is computing Twitter’s trending topics and sending them to the users’ browsers. An example of distributed RPCs is parallelizing search queries: a user invokes a function in Storm and as the (parallel) computation finishes, Storm delivers the results to the specified return address. Storm is extremely fast and scales well with the number of nodes.

Storm consists of a master node, similar to the master node in a MapReduce framework, and worker nodes. The master node is responsible for assigning jobs to the worker nodes. A worker node executes tasks by pulling data from a data source or a previous task in a series of tasks. Successful execution of a task is acknowledged to the master node. Acknowledged tuples are free to be pulled from the next task. Tasks may be stateful, but the state is kept in main memory and is not replicated. State may be written to a database as a side effect of a task.

With respect to the requirements of a real-time analytics system, Storm has the following characteristics. (1) Storm updates data in real time by actively pulling data from a data source or previous tasks. (2) The programming model consists of bolts and spouts. A bolt is a single-step stream transformation function that, for each input tuple, produces a set of output tuples. To aggregate data, state objects (i.e., aggregate values) must be kept explicitly within a bolt. In Limmat, map and reduce’ functions are stateless. Limmat’s execution framework stores and replicates aggregate values, and inputs aggregate values to reduce’ functions whenever these are called. A spout is a mediator between an external data source and a bolt. Furthermore, a topology defines how bolts and spouts are connected and to which extent bolts are synchronized. Depending on the synchronization setting in the topology, bolts can emulate map or reduce’ tasks. (3) Storm is fault-tolerant in that is replays tuples that were not acknowledged. Yet, all state that is kept within a bolt is lost in case of a node failure. Thus, Storm violates the fault-tolerance requirement. Storing the state in a database does not help, because writing to a database from within a bolt is a side effect and not part of any transaction. Thus, there would be inconsistencies in case of node failures. (4) The system scales well by partitioning the execution of bolts on multiple nodes. (5) Storm acknowledges every result exactly once.

With respect to the design space options, the authors of Storm made the following decisions. There is no replication of any data; that is, Storm will lose data in case of node failures. Storage (local state in main memory) and computation are combined. There is no support for hotspots. Transactions (i.e., the acknowledgements of results) are on a per-task basis.

Recently, iterative MapReduce engines gained focus of interest [22, 10, 121, 87, 78, 81]. All of this work has in common that they optimize the execution of
batch-oriented MapReduce jobs using iteration. HaLoop [22] processes batch-oriented MapReduce tasks iteratively until a user-defined fixpoint condition is met, thereby caching data to increase performance. Incoop [10] tries to optimize iterative executions of MapReduce by re-computing only those aggregate values, for which the input has changed; without the need to modify the MapReduce programming model. PrIter [121] allows to prioritize certain steps of a iterative computation for faster convergence. Ciel [87] dynamically extends a data flow depending on the data, enabling support for data-dependent iterative algorithms. The stateful groupwise operator by Logothetis et al. [78] allows to process small bulks of data iteratively. Finally, with Dryad [62] extending MapReduce’ control flow to directed acyclic graphs, Naiad [81] promises to add mechanisms for iteration (i.e., looping constructs) and incremental updates to Dryad. All of this work motivates the need for iterative computation and incremental updates to data. We build on this motivation to study the requirements and design of a real-time analytics system, which processes data in a MapReduce-like fashion as soon as new data arrives.

Stream processing systems (e.g., [26, 71, 1, 11, 88]) are closely related to Limmat. These systems process a set of input tuples (the input stream) via continuous queries to produce output tuples (the output stream). Typically, stream processing systems have a rich programming model (a streaming variant of SQL) including window semantics. Stream processing systems ensure in-order processing of data. NiagaraCQ [28] and TelegraphCQ [26] are early stream processing systems that run on a single node. Borealis [1] and IBM InfoSphere Streams [11] distribute the computation across multiple nodes by partitioning the computation by query operator. This results is limited scalability: A continuous query cannot be partitioned across more nodes than there are query operators. By relaxing the in-order processing requirement, Truviso [71] is able to partition the query processing by data, which leads to better scalability with the number of nodes of the system. Yahoo! recently developed the S4 streaming system [88]. Similar to Limmat, it provides a simple, MapReduce-like programming model. But S4 does not provide fault tolerance.

Limmat, in fact, can be seen as a stream processing system. Limmat processes a stream of input tuples and produces a set of aggregate values. Aggregate values are updated as new input tuples arrive and can be queried by the user. However, Limmat differs from stream processing systems with respect to the programming model and execution model. The MapReduce-like programming model of Limmat is much simpler that of streaming systems. There is no built-in notion of windows. Windowing functionality must be implemented by the application developer using map and reduce’ functions (e.g., by attaching timestamps to input tuples and
keeping a list of “buckets” of tuples per hour). A detailed study of complexity and efficiency of programming models of stream processing systems is out of the scope of this thesis. However, the simpler programming model of Limmat allows to partition the computation by key (i.e., aggregate values) across the nodes of the system, and thus Limmat scales with the number of nodes. Limmat does not ensure in-order processing of data.

4.6 Limmat

In this section, we describe Limmat. Limmat is a real-time analytics system that is scalable, fault-tolerant, and guarantees strong consistency, in particular exactly-once semantics. By implementing exactly-once semantics, Limmat essentially extends Percolator; and, by implementing fault tolerance, Limmat extends Storm. These additional guarantees are traded in for performance; but, as we will show in the next section, Limmat still shows good performance characteristics. As programming model, Limmat allows to express analytical tasks using the incremental map/reduce’ programming model of Section 4.3.

Limmat is built on top of a key-value store. A key-value store serves as a good basis because it naturally provides scalability (by partitioning the data) and, to a certain extend, fault tolerance (by, e.g., using a quorum-based consistency protocol). We extend the key-value store with queues and workers for task execution, two-phase commit for exactly-once semantics, and synchronization for consistency. In the following we will explain the details of these extensions and describe the execution of one map/reduce’ step.

4.6.1 Assumptions

We make the following assumptions about the underlying key-value store, which are required to implement Limmat.

1. The key-value store has an interface to read a value $v$ for a given table $tb$ and key $k$.

2. Any node of can be contacted to read a value or write a tuple.

3. Each node of the key-value store has a reliable, persistent storage.

4. The key-value store has a quorum-based replication protocol. The $(N, R, W)$ configuration is $R = \lfloor \frac{N}{2} \rfloor + 1$ and $W = \lfloor \frac{N}{2} \rfloor + 1$. 

70
5. All nodes of the key-value store know the definitions of all map/reduce jobs.

Data is inserted into Limmat as a sequence of tuples \((tb, k, v)\) where \(tb\) is a table, \(k\) is a key, and \(v\) is the value associated with the key. Each tuple is written to the storage. Writing a tuple \((tb, k, v)\) means storing the key-value pair \((k, v)\) in table \(tb\) of the key-value store. In Limmat, data is also shipped between nodes as tuples \((tb, k, v)\).

We call the component of the key-value store that replicates data replication subsystem and the component that stores data storage subsystem.

Persistent storage is needed for fail-recover semantics. In our system, every operation is written to the persistent storage using write-ahead logging. When a node recovers from a failure, it will read the log from the persistent storage and continue with the state it was in before the failure.

In the quorum-based replication protocol, we assume a \((N, R, W)\) configuration with \(R = \left\lfloor \frac{N}{2} \right\rfloor + 1\) and \(W = \left\lfloor \frac{N}{2} \right\rfloor + 1\). In other words, data can be read and written as long as the majority of nodes of a replication group are available.

These requirements are supported by several key-value stores; for example, by Apache Cassandra [74], which we use as the basis to implement Limmat.

### 4.6.2 Additional Mechanisms

Based on these assumptions, we implement additional mechanisms on top of the key-value store.

**Queues and Workers.** We extend each node of the key-value store with queues and worker threads. There are two queues for each map/reduce: one queue that buffers map tasks, and one queue that buffers reduce tasks. Tasks are tuples \((k, v, f)\) where \(k\) is a key, \(v\) is a value, and \(f\) is the map or reduce function to be called (i.e., the task is executed). Tasks are durably stored and removed from the queues (using distributed transactions). There are multiple worker threads per queue. For each queue, worker threads drain the queue and execute tasks in parallel. One advantage of using queues is the ability of handling bursts of input data per second (as in the previous chapter). Furthermore, the sizes of queues of a node allows a rough estimation of the load of a node.

**Two-phase Commit.** We implement the two-phase commit protocol described in [100] to execute distributed transactions. Distributed transactions are needed for atomic processing of tasks. That is, a worker thread fetches a task from a queue, executes the task, and forwards the results to another node or stores them in a table of the key-value store; and all of that happens atomically, all or nothing.
Thus, we ensure that each task is successfully executed exactly once. As part of two-phase commit, all operations are persistently written to the storage using write-ahead logging [100].

**Synchronization.** We add routing and locks to synchronize the execution of reduce tasks. Reduce tasks need to be synchronized because several tasks can potentially update the same aggregate value in parallel leading to inconsistent data. Synchronization is realized in two steps: by routing all key-value pairs output by a map function with the same key to a single node that executes the corresponding reduce function, and by synchronizing the execution of the reduce function within a node using locks. Routing is implemented reusing the key-value store’s partitioning strategy—the primary replica storing the corresponding aggregate value is chosen. In case this primary replica fails, the output of map is routed to the second replica (and so forth). In this case, synchronization between nodes is required. This is achieved using fixed-length leases [45]. Limmat has a central master node that serves as lease manager. Database nodes acquire leases from this manager. Leases expire after a fixed period of time (30 seconds in our case) and need to be renewed. Only the node that obtained the lease for the corresponding key range is allowed to execute reduce tasks. In the normal case, the primary replica node already acquired the lease for its key range and can process reduce tasks for all keys routed to the node. In the failure case, when key-value pairs are routed to another replica, this replica must first acquire the lease for the key range before starting to process these pairs.

### 4.6.3 Architecture

Figure 4.6 illustrates the architecture of one node of Limmat. On top of the existing components of a node of the key-value store, we add an *insert handler*, a *map handler*, a *reduce handler*, queues, and *worker threads*. The insert handler handles the insertion of input tuples and communicates with the replication subsystem. The map handler buffers tasks in the (map) queues. Similarly, the reduce handler buffers tasks in the (reduce) queues. Map queues are drained by worker threads (denoted by *MT* in Figure 4.6) that execute map tasks. Reduce queues are drained by worker threads (denoted by *RT*) that execute reduce tasks. The replication subsystem forwards writes of tuples to the storage subsystem. Every communication between handlers is acknowledged and committed as defined by the two-phase commit protocol. (Acknowledgement and commit messages inside the node are not shown in Figure 4.6).

The input to the node is either a write request of an input tuple or a read request. The write request arrives at the insert handler. The write request is answered with
an acknowledgment (ACK). The read request arrives at the original key-value store component and consists of a table \( tb \) and key \( k \). As a result, the value for key \( k \) in table \( tb \) is returned. Furthermore, the replication subsystem of one node of Limmat communicates with replication subsystems of all other nodes of Limmat.

We do not the *gossiping subsystem* of key-value store to communicate with any handler (as was the case in the previous chapter). As every step of the execution of a map/reduce' job is a distributed transaction, a node failure will result in a timeout and, thus, in the abort of the transaction. Explicit communication with the gossiping subsystem is not needed.

### 4.6.4 Execution of map/reduce'

Given the mentioned extensions and the architecture, we now explain how an insertion of an input tuple and one map/reduce' step is executed in Limmat. The detailed steps of the execution are shown in Figure 4.7. The execution consists of three steps: (1) the insertion of an input tuple, (2) the execution of a map function, and (3) the execution of one or more reduce' functions. If multiple map/reduce' jobs are defined on one input table, an input tuple triggers the execution
of multiple map functions. Successive map/reduce steps use as input the results produced by earlier steps.

In the following, we assume that the map/reduce job consists of a map function $f_m$, a reduce function $f_r$, an input table $t_{in}$, and an output table $t_{out}$.

**Inserting Data.** Whenever a new tuple $(t_{in}, k, v)$ is inserted into the system, the insert handler of the contacted node runs a distributed transaction using two-phase commit. The insert handler is the coordinator of the distributed transaction. The transaction consists of three steps: First, the input tuple is replicated (i.e., stored) to the majority of the nodes of the replication group. Second, one node among the replication group is elected to be the coordinator to execute the map task. The insert handler sends an election bit to elect a node. Third, the map handler of the elected coordinator node puts the corresponding map task $(k, v, f_m)$ into the map queue. As a result of this transaction, the input tuple has been stored in a replicated fashion, and one node buffers the map task. In Figure 4.7, the contacted node $N_0$ replicates the tuple $t_{in}, k_1, v_1$ and elects node $N_1$ as coordinator to execute the map task. The map queue of node $N_1$ contains the map task (denoted as $m$).

**Executing map.** Eventually, a worker thread $MT$ executes the map task $(k, v, f_m)$. The execution of the map task is also a distributed transaction. Worker thread $MT$ is the coordinator of the distributed transaction. The transaction consists of four steps: First, the map task is removed from the queue. Second, the map function $f_m$ is called with parameters $k$ and $v$. Third, each key-value pair $(k_m, v_m)$
output by the map function is written to the corresponding reduce' node\textsuperscript{12} as a tuple \((t_{vt}^{r}, k_{m}, v_{m})\), where \(t_{vt}^{r}\) is a virtual intermediate table. The virtual intermediate table is a not a real table, but only a placeholder such that the reduce' handler at the contacted node knows which function \(f_{r}\) to call. Fourth, at each contacted node, the reduce' handler puts a reduce' task \((k_{m}, v_{m}, f_{r})\) into the queue. In Figure 4.7, node \(N_{1}\) executes the map task and writes \((t_{vt}^{r}, k_{2}, v_{2})\) to the responsible node \(N_{2}\). The queue of \(N_{2}\) contains the reduce' task (denoted as \(r'\)). Note that a tuple \((t_{vt}^{r}, k_{m}, v_{m})\) output by a map function is not replicated, but only forwarded to one node, defined by key \(k_{m}\). In Figure 4.8, two map tasks are executed on different nodes \((N_{1} \text{ and } N_{4})\). Output tuples of map tasks with the same key \(((t_{vt}^{r}, k_{2}, v_{2}) \text{ and } (t_{vt}^{r}, k_{2}, v_{4}))\) are routed to the same node \((N_{2})\) to ensure synchronization of the execution of reduce'. (Replication is not shown in Figure 4.8.)

**Executing reduce'.** Eventually, a worker thread \(RT\) executes the reduce' task \((k_{m}, v_{m}, f_{r})\). The execution of the task only starts if the node on which worker thread \(RT\) is running is in possession of the lease for the key range that includes key \(k_{m}\). Furthermore, locally, worker thread \(RT\) must acquire a lock for \(k_{m}\). The execution of the reduce' task is again a distributed transaction. Worker thread \(RT\) is the coordinator of the transaction. The transaction consists of the following three steps: First, the worker thread reads the old aggregate value \(agg_{old}\) for

\textsuperscript{12}As discussed, if this node is not available, the next node in the replication group will be chosen.
key $k_m$ and table $t_{out}$ from the majority of nodes of the replication group. (The worker thread knows parameter $t_{out}$ because of function $f_r$ and the fact that all nodes know all definitions of map/reduce' jobs.) Second, the worker thread calls function $f_r$ with parameters $k_m$, $v_m$ and $agg_{old}$. It yields a new key-value pair ($k_m$, $agg_{new}$). Third, tuple $(t_{out}, k_m, agg_{new})$ is replicated to the majority of nodes of the replication group. In Figure 4.7, node $N_2$ executes the reduce' task and replicates its results.

If a subsequent map/reduce' task is defined on input table $t_{out}$, worker thread $RT$ elects one of the replica nodes it replicates tuple $(t_{out}, k_m, agg_{new})$ to as coordinator to execute the next map task. The election is also part of the transaction of worker thread $RT$.

### 4.6.5 Discussion

With respect to the requirements of a real-time analytics system, Limmat has the following properties.

**Processing Data in Real Time.** Limmat updates data incrementally, in real time. Whenever new data is inserted into Limmat, it triggers the execution of map and reduce' tasks. Tasks are executed as fast as possible.

**Simple Programming Model.** Limmat provides the simple, MapReduce-like programming model presented in Section 4.3.

**Fault Tolerance.** Limmat ensures data availability and non-blocking execution in case of node failures. Data availability is ensured by replicating data to multiple nodes. As long as a majority of nodes of the replication group are alive, data can be read and written. Non-blocking execution is closely coupled with data availability. Limmat continues to make progress in case of node failures as long as a majority of nodes of the replication group are alive. One of these alive nodes will execute tasks. Only the majority of nodes of the replication group is needed to make progress. Our system blocks if the majority is lost, for as long as it takes until a majority is reached again. Tasks, which were (persistently) buffered in the queue when the node went down, will be executed whenever the node recovers from the failure. If the lease manager dies, the system stops working until the lease manager recovers. This single point of failure is similar to the master node in the original MapReduce framework [32]. A solution is to make the lease manager fault-tolerant by replicating its state (as, e.g., done in the Chubby distributed lock service [23]).

**Scalability.** Limmat scales with the number of nodes because task execution is distributed across all nodes of the key-value store.
Distributed transactions do not prevent scalability because the number of nodes involved in a single distributed transaction is small, and does not depend on the size of the key-value store. The number of nodes involved in a transaction executing a reduce task is constant. It is determined by the replication factor (= 3 in Figure 4.7) plus the node executing the reduce task. The number of nodes involved in a transaction executing a map task depends on the number of tuples output by the map task. Output tuples of a map task are not replicated. Typically, because one map task operates on a single input, the number of output pairs is small. If there are too many output tuples produced by a map task, and therefore too many nodes involved in a distributed transaction, a solution is to split the map transaction into execution and forwarding. The transaction executing map would commit all output tuples to a local buffer. And multiple small transactions would forward the contents of this buffer to the respective nodes. In summary, each distributed transaction in Limmat involves only a small number of nodes, independent of the size of the key-value store; and therefore, distributed transactions do not prevent scalability.

**Strong consistency.** Limmat provides exactly-once semantics by atomically executing tasks and synchronizes the execution of reduce tasks. The execution of an insert of an input tuple and each part of the execution of a map/reduce step is a distributed transaction. In case of any failure, the transaction is rolled back. In particular, map and reduce tasks remain in the queue and are executed again until their successful completion.

With respect to the design space options of Table 4.1, we made the following decisions. (1) Externally visible data (input and result data) is replicated. This ensures availability and non-blocking execution. (2) Intermediate data is not replicated, because we assume nodes with persistent storage. (3) Computation and storage are tightly coupled. The computation is performed on the nodes that store the data. (4) Hotspots are not supported. Adding support for hotspots is left for future work. (5) Transactions are on a per-task basis and do not span multiple tasks.

### 4.7 Experiments and Results

In this section, we study the performance characteristics of Limmat, and compare it to Twitter Storm and different variants of Limmat itself. We are able to configure essential parameters of Limmat, which allows us to compare Limmat’s strong-consistency configuration to variants of Limmat, which have the same characteristics as Google Percolator and Twitter Storm. Furthermore, we compare
Limmat to Hadoop, a batch-oriented MapReduce implementation. Last, we study the trade-offs of the design space options.

### 4.7.1 Software and Hardware Used

**Limmat.** We implemented Limmat as described in Section 4.6. Limmat was implemented in Java 1.6 borrowing parts of the messaging, partitioning, and replication source code of Cassandra [74]. We implemented the queues using Java’s ConcurrentLinkedQueue, ConcurrentHashMap, and AtomicLong classes as underlying data structures. Further, we implemented worker threads, which drain the queues and execute tasks, using standard Java threads. Each map queue is drained by 16 worker threads. Each reduce queue is drained by 32 worker threads. Each node of Limmat has 8 TCP connections to each other node of Limmat, resembling a fully connected graph. Two-phase commit is implemented following the description in [100]. Consequently, each operation in Limmat (i.e., each prepare, abort, and commit; but no read) is persistently logged to the hard disk of a node using write-ahead logging. Each write to the write-ahead log is immediately flushed to the hard disk. Synchronization within a node is implemented using Java’s ConcurrentHashMap, which stores locks identified by key. The replication factor is set to 3, the read quorum to 2, and the write quorum to 2.

Limmat allows to vary essential parameters, such as the replication factor, and enabling or disabling two-phase commit. This makes it possible to compare different variants of Limmat. The variants of Limmat, which we compare, are the following:

- **Limmat (strong consistency):** Two-phase commit is *enabled*. Externally visible data is replicated. Intermediate data is not replicated.
- **Limmat (Percolator):** Two-phase commit is *disabled*. Externally visible data is replicated. Intermediate data is not replicated.
- **Limmat (Storm):** Two-phase commit is *disabled*. Externally visible data and intermediate data are *not* replicated.

Limmat (strong consistency) is the configuration discussed in Section 4.6. In the Limmat (Percolator) configuration, there is no two-phase commit. Intermediate data is not replicated because in Google Percolator there is no intermediate data, which is written (and replicated) to the database (see Section 4.5). Limmat (Storm) does not replicate data at all.

**Twitter Storm.** We used the open-source version of Twitter Storm. Storm has been open-sourced in September 2011. Since then, it has been actively developed...
at a rapid pace. The version of Twitter Storm we used in the experiments was 0.5.3. Storm was configured as suggested by its developers; that is, using one master node (called Nimbus), one Zookeeper node, and multiple worker nodes. The number of worker nodes depends on the experiment. On each worker node, we allowed 16 threads for the bolt (see Section 4.5) that ran a map task and 32 threads for the bolt that ran a reduce task. Furthermore, we used 4 spouts per node that pull data from the data source. 4 spouts were enough to make sure that pulling data from the data source is not a bottleneck.

**Hadoop.** We used one of the latest versions of Hadoop, Version 0.20.203.0. We configured Hadoop using the default configuration, but set the replication factor of the Hadoop filesystem (HDFS) to 3. This matches the replication factor of Limmat for a fair comparison of the two systems. Before the start of Hadoop, all data was already inserted into HDFS. The time to initially insert the data into HDFS was not measured when evaluating Hadoop.

**Hardware.** All experiments were carried out using 4 machines. The machines are equipped with an Intel Xeon 3360 2.8GHz processor and 8 GB RAM running Ubuntu Linux 10.04. In the scalability experiment, we increased the number of machines up to 16 machines, 11 of which are equipped with an AMD Opteron 2376 2.3GHz processor and 16 GB RAM. In the fault tolerance experiment we started Limmat and Twitter Storm on 16 machines and gradually shut down machines, one by one.

One additional machine was used to serve as the data source. The data source inserts data into Limmat, and allows Twitter Storm to pull data. The data source can be configured to insert data as fast as possible (to saturate the system), or issue data at a certain rate. Another machine was used to serve as lease manager, and, for the experiment involving coarse-grained transactions, as lock manager.

### 4.7.2 Benchmark

Our experiments were performed using up to 8 GB of real data from Twitter. We recorded 3.6 million tweets (≈ 8 GB) using the Twitter Streaming API [111]. Each tweet is a serialized JSON string with 19 atomic values such as tweet id, creation date, and tweet text; and 2 nested elements for the tweet author and special entities (hashtags, URLs, and mentions). The average size of a single tweet, serialized as JSON, is 2.3 kB.

As analytical task, we used the word count application as described in the original MapReduce paper [32] and in Section 2.2. The task is to count the number of occurrences of each word in all tweets. We count only words with alphabetic characters; no numbers, special UTF-8 symbols, URLs, or compounds of letters.
and numbers. The application has been implemented in Limmat using the map/reduce implementation shown in Section 4.3. Additionally, within the map task, the serialized JSON string has to be parsed to extract the text of the tweet. Twitter Storm and Hadoop come with examples of the same word count application. We extended these examples to be able to parse JSON. In all systems, we used the same JSON parsing library org.json.jar, for a fair comparison.

4.7.3 Real-time Analytics

In the following experiments, we evaluate the requirement of Limmat and Twitter Storm to process data in real time.

Single Inputs

In the first experiment we measure the completion time of a map/reduce job when updating counters for single, sequential inputs. That is, we measure how long it takes to split a tweet into words and increment the respective counters of the words. As job completion time, we take the time between when a tweet is input until the last counter of a word has been incremented. The time is measured at the data source machine. For that, each reduce task pings the data source machine after it has incremented a counter and committed the transaction. The last ping marks the completion of the job. Before the start of the experiment, 2 GB of data (900,000 tweets) have already been processed. Tweets are input sequentially, but multiple reduce tasks may run in parallel within the system. The experiment consisted of 4 nodes.

Table 4.2 shows the job completion times for Twitter Storm and different configurations of Limmat. We make the following observations: (1) Both systems update data in the milliseconds range, in real time. (2) Twitter Storm processes data the fastest, but the Limmat (Storm) configuration comes close. Both approaches do not replicate data and are therefore the fastest. (3) The Limmat (Percolator) configuration is about two times slower than the previous approaches because it replicates results (i.e., the word counters). (4) Limmat (strong consistency) is even slower because of the two-phase commit protocol to ensure exactly-once semantics. However, this time penalty of two-phase commit is still acceptable.

13http://json.org/java/
Table 4.2: Job completion time for Twitter Storm and different configurations of Limmat. (Sequential input, 4 nodes)

<table>
<thead>
<tr>
<th>System</th>
<th>Completion time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
</tr>
<tr>
<td>Twitter Storm</td>
<td>1.10</td>
</tr>
<tr>
<td>Limmat</td>
<td>1.70</td>
</tr>
<tr>
<td>- Storm</td>
<td>2.34</td>
</tr>
<tr>
<td>- Percolator</td>
<td>3.33</td>
</tr>
<tr>
<td>- strong consistency</td>
<td></td>
</tr>
</tbody>
</table>

**Burst of Inputs**

In the second experiment, we measure how long each system needs to process a burst of 2 MB of data. The data is input at once, concurrently. Thus, map and reduce’ tasks run in parallel and try to process the data as fast as possible. Again, before the start of the experiment, 2 GB of data (900,000 tweets) have been already processed. The size of the burst is 2 MB (900 tweets). The experiment consisted of 4 nodes.

Figure 4.9 shows the results of this experiment. It shows the number of concurrently running map and reduce’ tasks during 50-millisecond samples. We make the following observations: (1) Twitter Storm and Limmat (Storm) allow a high degree of parallelism of up to 2000 reduce’ tasks that run concurrently (i.e., within a 50-millisecond interval). Limmat (Percolator) is able to schedule around 1000 reduce’ tasks in parallel, and Limmat (strong consistency) roughly 600 reduce’ tasks. (2) It follows that Limmat (Storm) takes only 200 ms to analyze the 2 MB burst of data, whereas Limmat (Percolator) takes 350 ms, and Limmat (strong consistency) takes 550 ms. (3) An interesting observation is that Twitter Storm schedules subsequent tasks only after it has gathered some results from previous tasks, shipping data in batches. Therefore, Twitter Storm takes 400 ms to process the burst. Limmat sends data without batching and triggers tasks as fast as possible; thus, many reduce’ tasks are executed even within the first 50 ms.

**Burst of Inputs, Vary Existing Data**

In the third experiment (which was shortly discussed in the introduction already), we vary the size of the existing data; that is, the data that has already been processed before we insert a burst of 2 MB of new data. Before the start of the experiment, we process up to 8 GB of data and then measure the time it takes to
update this data with 2MB of new data. The size of the new data is constant. In this experiment, we compare Limmat (strong consistency) to Hadoop. In order to update the existing data with new data, Hadoop has to re-compute all of the data, existing and new, to produce new results. Again, we used 4 nodes in this experiment.

The results are shown in Figure 4.1, in the introduction of this chapter. As already discussed, there are two important observations: (1) Hadoop needs minutes instead of milliseconds to produce up-to-date results. This is because Hadoop needs to re-compute all of the data, whereas Limmat only updates the necessary counters. (2) The completion time of Hadoop depends linearly on the size of the existing
data, whereas for Limmat it depends on the size of the new data. The completion time of Limmat remains constant, at around 600 ms (which was also shown in Figure 4.9(d) for 2 GB of existing data).

4.7.4 Scalability

In the next experiment we evaluate the requirement of a real-time analytics system to scale with the number of nodes. We continuously insert tweets into the system and measure the throughput the system is able sustain. In this experiment, the system is saturated.

Figure 4.10 shows the throughput of Twitter Storm and the different configurations of Limmat when increasing the number of nodes of the system. We observe the following: (1) All systems scale with the number of nodes of the system—the more nodes are added to the system, the more messages per second can be processed. (2) Twitter Storm and Limmat (Storm) perform the best because they do not replicate data. Twitter Storm scales even better than Limmat (Storm) because it optimizes the computation by using batching. All variants of Limmat do not use batching because for Limmat (strong consistency) we did not see any performance benefit, and Limmat (Storm) and Limmat (Percolator) are only variants of Limmat (strong consistency). (3) Limmat (Percolator) has higher throughput than Limmat (strong consistency) because it does not use two-phase commit and can process more messages concurrently.

Limmat’s distributed transactions do not prevent scalability. As explained in Section 4.6.5, transactions involve only a small number of nodes. Therefore, more
and more transactions can run in parallel as the number of nodes of the system increases. This is an important result as it shows that, even when using two-phase commit, Limmat scales with the number of nodes.

4.7.5 Fault Tolerance and Strong Consistency

We now evaluate the last requirements of real-time analytics systems, fault tolerance and strong consistency. We evaluate fault tolerance by starting both systems, Limmat and Storm, with 16 nodes. We then gradually force node failures, one by one. We wait until the system stabilizes until forcing the next node failure. The experiment continues until a system stops analyzing data. We measure the percentage of available data a user from the outside is able to read, as well as the throughput of each system. Here, we throttle the input rate to 2000 tweets per second and measure how many of these are actually processed. That is, we are interested if all input data can be analyzed in the event of node failures. We restrict the analyzed systems to Twitter Storm and Limmat (strong consistency), but will discuss how Limmat (Storm) and Limmat (Percolator) would behave.

Figure 4.11(a) shows data availability for Twitter Storm and Limmat (strong consistency). Storm does not replicate data and, thus, from the first node failure on data is not 100% available. Limmat (strong consistency), instead, guarantees 100% availability for up to 5 node failures (i.e., 11 or more nodes that are alive). After that, replication groups in Limmat begin to collapse and Limmat cannot guarantee data availability. The data is still persisted on the nodes’ hard disks, but cannot be read from a user outside of the system. We did not implement dynamically re-adjustable replication groups in Limmat; therefore, data becomes unavailable.

Figure 4.11(b) shows the throughput of the two systems when inserting 2000 messages per second. Ideally, the systems should process these message without blocking because of node failures. Both, Storm and Limmat, do so. Yet, there is a major difference. Twitter Storm is able to process 2000 messages per second in case of node failures, but as soon as one node goes down, Storm begins to output wrong results. The execution framework of Twitter Storm routes an incoming message to a different node in case the originally responsible node failed. The new node does not contain the previous counter value of the failed node, and starts counting words from 0 again. Thus, Storm produces wrong results. Limmat, instead, always produces correct results in all cases, due to data replication and two-phase commit. For example, when the first node went down, Limmat aborted and successfully restarted 6 transactions that were affected by the node failure. From 10 alive nodes on, Limmat begins to analyze only parts of the messages. The ma-
Figure 4.11: Fault-tolerance. Data availability (a) and non-blocking execution (b) when forcing node failures. Starting with 16 alive nodes, killing nodes one by one.

Majority of the messages cannot be analyzed because the responsible nodes are not available to read and write data. Requests are queued inside Limmat until some of the nodes become available again. With only 5 alive nodes, Limmat cannot process data at all.

Limmat (Storm) would show the same data availability curve as Twitter Storm in Figure 4.11(a). Limmat (Percolator) would be similar to Limmat (strong consistency). Yet, Limmat (Percolator) would not abort and restart transactions as Limmat (strong consistency), but tasks would be stopped without a restart. For example, when the first node failed in Figure 4.11(b), 6 tasks would stop in the middle of their execution without a restart. Limmat (Percolator) only guarantees at-most-once semantics.

We conclude that both, Twitter Storm and Limmat (Percolator), cannot process data correctly in the event of node failures. Limmat (strong consistency), on
Table 4.3: Completion time of a map/reduce job for Limmat (strong consistency) when replicating intermediate data or using coarse-grained transactions. (Sequential input, 4 nodes)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Completion time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
</tr>
<tr>
<td>Limmat (strong consistency)</td>
<td>3.33</td>
</tr>
<tr>
<td>Replicating intermediate data</td>
<td>5.41</td>
</tr>
<tr>
<td>Coarse-grained transactions</td>
<td>120</td>
</tr>
</tbody>
</table>

the other hand, processes data correctly, and provides data availability and non-blocking execution.

Note that if two nodes of the same replication group fail, Limmat cannot process data for that replication group. Figure 4.11 shows the best-case scenario in which nodes of different replication groups fail. Only when the 6th node failed, the first replication group collapses. To circumvent the collapsing of replication groups, dynamically re-adjustable replication groups (e.g., based on hinted hand-offs [34] or re-replication [6]) would need to be implemented.

4.7.6 Design Space Options

Replicating Input and Result Data. Replicating externally visible data is crucial to guarantee data availability. This has been shown in Figure 4.11(a). The cost of replication is shown in Figure 4.10, where both configurations of Limmat, which replicate data, can process roughly 30% of the messages that Twitter Storm and Limmat (Storm) can process. This difference can be explained by the replication factor, which was set to 3.

Replicating Intermediate Data. We measured the job completion time and throughput for Limmat (strong consistency) when replicating intermediate data. The benchmark setup for the job completion time experiment was the same as in Section 4.7.3 (Single Inputs). The setup of the throughput experiment was the same as in Section 4.7.3 (Scalability), but with only 4 nodes. The results are shown in tables 4.3 and 4.4. Replication of intermediate data yields a slightly higher job completion time, but a still acceptable one. Yet, the throughput drops significantly by 66% in comparison to Limmat (strong consistency). Therefore, we chose to implement Limmat without replication of intermediate data.

Distribution of Computation and Storage. The experimental evaluation of distribution of computation and storage has been studied in Chapter 3. The
findings were that the combined approach uses 35% less network resources and 15% fewer machines to perform and scale equally well as the separated approach.

**Support for Hotspots.** An experimental study of adding support for hotspots is out of the scope of this thesis and left for future work. Limmat, currently, does not support hotspots and will buffer requests in queues if a node gets overloaded.

**Granularity of Transactions.** We evaluated the trade-off of granularity of transactions. We implemented coarse-grained transactions in place of the fine-grained transactions in Limmat (strong consistency). Therefore, we setup a lock manager node, which can be contacted to acquire global locks. Also, we implemented a new map/reduce job, which counts the number of occurrences of words in tweets as follows: The map function parses a tweet and splits the tweet text into words. It contacts the lock manager to acquire global locks for all words. It then increments all counters by reading and writing to the key-value store. Last, it releases all locks. To avoid deadlocks, if one of the locks cannot be acquired, all locks are released and the map task tries to acquire the locks again. There is no reduce function; all processing happens within the map function. Map tasks are executed in parallel across 4 nodes of the cluster without synchronization (i.e., nodes do not acquire leases). The only means of synchronization is the lock manager.

The results are shown in tables 4.3 and 4.4. The median job completion time for a task with coarse-grained transactions is 36 times higher than for fine-grained transactions (i.e., Limmat with strong consistency). The throughput drops to 23 processed tweets per second. We conclude that coarse-grained transactions, which involve the execution of a map task and multiple reduce tasks, are not feasible. For each task execution, locks must be acquired and released; whereas Limmat uses leases (which are acquired only every 30 seconds) and uses locks only locally, which is efficient.

### Table 4.4: Number of processed messages per second for Limmat (strong consistency), replicating intermediate data, and coarse-grained transactions. (4 nodes)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Throughput [msg/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limmat (strong consistency)</td>
<td>3520</td>
</tr>
<tr>
<td>Replicating intermediate data</td>
<td>1176</td>
</tr>
<tr>
<td>Coarse-grained transactions</td>
<td>23</td>
</tr>
</tbody>
</table>

87


4.8 Conclusion

In this chapter, we discussed the requirements and design space options of a real-time analytics system. For such a system, updating data incrementally is crucial to achieving good data freshness. Furthermore, a real-time analytics system should have a simple, yet expressive, programming model; scale with number of nodes; tolerate node failures; and provide strong consistency. Given these requirements, we studied important design space options: (1) replication of externally visible data (input and result data) is crucial to providing data availability; (2) replication of intermediate data allows better data freshness, but significantly impacts performance; (3) combining computation and storage is beneficial (which has been discussed in detail in the previous chapter); and (4) distributed transactions should be fine-grained to allow many concurrent transactions.

Based on these results, we presented Limmat, a real-time analytics system. Limmat extends Google Percolator by providing exactly-once semantics of task execution, and Twitter Storm by providing strong consistency in the event of node failures. These advantages are traded in for performance. But, as shown in Section 4.7, the performance of Limmat is still acceptable and, in spite of distributed transactions, Limmat scales linearly with the number of nodes.

Finally, we motivated the need for further research concerning the support for hotspots. We believe that automatic handling of hotspots is a crucial factor for building even more robust real-time analytics systems. Yet, except the work in [95], we are not aware of any advances in this direction.
Chapter 5

The Newspaper Optimization Application

To demonstrate Limmat, we developed an application to optimize online newspapers using social media. Today, in more than 40% of all cases, news are shared via social networks such as Twitter, Facebook, and YouTube [106]. Using real-time feedback from these media, news companies can understand the impact of recently published news. This, for example, allows them to reorganize their online newspapers placing most important stories first and, thus, attract more readers. To support this use case, our application uses Limmat to analyze shared news in social media in real time.

This application was part of our demonstration of Limmat at the 2011 VLDB conference [46].

In the following, we describe the application and its implementation, and present demonstration scenarios.

5.1 Application

Our application is a service for optimizing online newspapers based on information posted by people on Twitter. In Twitter, many tweets contain links to news articles and blog posts [106]. Our application analyzes these links and sorts them by how many times these links have been shared. Using this information, news companies can monitor the popularity of published articles and reorganize the front pages of their online newspapers accordingly. Hopefully, this will attract more readers to their newspapers.
Our application runs on top of Limmat. The input to our application is a sequence of tweets from Twitter. The output of our application is a sorted list of URLs grouped by domain. The list of URLs is sorted based on the number of occurrences of URLs in all tweets. Sorted lists are stored in Limmat and can be retrieved using read requests.

Figure 5.1 illustrates the use case. In Figure 5.1, the WSJ.com online newspaper (the online edition of the Wall Street Journal) publishes several articles. The second one of these articles gets tweeted by many people. That is, people share a link to this article on Twitter. Following this tweet, other people read this article and retweet this link. (Retweeting means to republish a friend’s tweet to your own friends.) The article is shared among many people on Twitter—it is popular. As our application keeps a sorted list of URLs for each domain, the URL of the popular article will be on top of the sorted list for the WSJ.com domain. To get feedback on popular articles, the WSJ.com online newspaper can retrieve the list for domain WSJ.com from Limmat. Using this feedback, the online newspaper can now reorganize its list of articles placing the popular article on top. Thus, the articles on the front page of the newspaper are ranked based on their popularity in social media.

In detail, our application consists of the following four steps: (1) Our application extracts links of tweets and expands them. Typically, links are abbreviated using URL shorteners such as bit.ly and first have to be expanded to obtain the original (long) URL. (2) The application counts the number of occurrences of each URL in all tweets. This is similar to the word count application discussed in sections 2.2 and 4.3, which counts the number of occurrences of words in documents. (3) URLs are then grouped by their domain. And (4), for each domain, the application keeps a sorted list of URLs based on the number of their occurrences.

We implemented this application on top of Limmat via two successive map/reduce jobs. The pseudo code of the application is shown in Figure 5.2. The jobs require three tables: tweets, urlCounts, and domain. The tweets table stores tweets, which
Figure 5.2: Pseudo code of our application.

are JSON objects serialized as strings, associated with a unique identifier. The urlCounts table stores counters for each URL. The domain table stores a sorted list of (url, count) pairs for each domain. The first map/reduce' job (Figure 5.2(a)) has input table tweets and output table urlCounts. The map function parses a tweet, extracts all URLs from the tweet, and expands these URLs. The reduce' function counts the number of occurrences of each URL. The second map/reduce' job (Figure 5.2(b)) has input table urlCounts and output table domain. The map function extracts the domain of the URL and outputs the domain and a (url, count) pair. The reduce' function takes as input the domain, the (url, count) pair, and the sorted list of (url, count) pairs for this domain. It removes the old pair with URL url from the list (if it exists), and adds the new pair (url, count) to the list. Then it sorts the list by the counts of the pairs and outputs the new list. As a result, a user can look up the most popular URLs for any domain by sending a read request to Limmat (e.g., a read request with table “domain” and key “WSJ.com”).

5.2 Demonstration Scenarios

We demonstrated Limmat and our application in three scenarios. Thereby, we used real data obtained from Twitter. Twitter allows to access a 1% sample of all current tweets in real time [111]. We used this data as is to demonstrate real-time analytics. To demonstrate the scalability of Limmat, we artificially scaled
this workload to the 100% Twitter workload (2000 messages per second [90]) using
data previously accumulated over seven days.

In the scenarios described below, we used a cluster of machines where each ma-
chine was equipped with an AMD Opteron 2.4 GHz CPU and 6 GB RAM running
Ubuntu Server 10.

**Scenario 1: Monitoring.** A user accessed our service through a standard web
browser. The start page displayed a text field in which the user could enter any
domain, for example *wsj.com*. For this domain, the service then displayed a sorted
list of URLs that were currently discussed on Twitter. The list was constantly
updated in the browser as new tweets were received, which demonstrated the real-
time character of our application. It allowed the user to see the most popular
articles of the Wall Street Journal online newspaper on Twitter at that time. The
user could reset the displayed list for the domain by clicking a reset button. When
the user clicked this button, all statistics were set to zero. In an instant though,
the list was populated again as new tweets were received.

**Scenario 2: Scalability.** To demonstrate the scalability with the number of
nodes of Limmat, we artificially scaled up the input stream to mimic a 100%
Twitter load. That is, we replayed data accumulated over seven days in a much
shorter time frame. We showed that Limmat was able to handle this load using
a cluster of 15 machines. The user was able to see that URL counters increased
much quicker with the scaled-up input stream.

**Scenario 3: Fault Tolerance.** To demonstrate non-blocking execution of map/
reduce′ jobs in case of node failures, the audience was able to pick any node of
Limmat that we would then manually shut down by killing the corresponding
Java process. To support this scenario, Limmat logs all fatal errors during task
execution. Using the Linux command `tail -f error.log` (which outputs all errors
to a console), we could show that no such errors occurred when shutting down a
node of Limmat. The audience was allowed to choose more nodes to get killed,
which at some point caused fatal errors to appear in the logs. This happened when
the majority of nodes of a replication group was down.
Chapter 6

Mapping Data to Queries

This chapter marks the start of the second part of this thesis. In this part, we study a new approach to data integration. Our approach, called Mapping Data to Queries, is radically different to existing approaches to data integration and outperforms these by orders of magnitudes in extreme cases.

The contents of this chapter are currently under submission as a journal paper. We also demonstrated MDQ at the 2010 VLDB conference [58].

6.1 Introduction

Data integration is one of the big challenges of modern information systems. The goal of data integration is to provide a familiar interface for a user to query and update data that originates in diverse data sources. Typically, mapping rules are developed to relate schemas of the various sources. Users then issue queries and updates according to one schema and these queries are automatically translated, via the mapping rules, into queries and updates on any mapped schema—or data is automatically transformed into the schema understood by the queries.

Many applications require the integration of large numbers of schemas. In industries such as health care and finance, organizations may have their own formats for important data such as patient records or accounts, leading to hundreds or thousands of different schemas for this data. And, this diversity of schemas is not going away; instead, Health Level 7 [56] and XBRL [117] focus on defining conventions for interoperability across schemas. Health Level 7, for example, does not dictate a specific schema for patient records; instead it gives each hospital and health insurance company the flexibility to define their own schemas and specifies
conventions on how to define these schemas so that different organizations can interoperate.

We envision a world of interoperability that is even more heterogeneous than today. A world where there are documents with mixes of schemas. That is, part of a document corresponds to one schema while other parts may correspond to other schemas. This is especially possible for semi-structured data formats, which are typically used to interchange information. Semi-structured data carries information about the schema within the document itself and, thus, may be composed of different parts which correspond to different schemas. For example, a health record of a patient may contain a history of findings from several hospitals. Each finding is stored in the schema of the respective hospital resulting in a health record with a mix of schemas.

There are two traditional approaches to data integration: transformations and query rewrite (or federation) [114]. Both approaches perform poorly when scaling with the number of schemas. The transformation approach, based on using mapping rules to transform the data and storing copies of the transformed data in a database (or data warehouse), performs poorly because all data is transformed independent of whether the data is needed to produce the query result, and because mixes of schemas lead to slow transformation programs that need to (recursively) consider all possible schema mixes. The query rewrite approach, based on using mapping rules to rewrite original queries into queries that can be executed on the heterogeneous data, performs poorly because all possible schemas and mixes have to be included in the rewritten queries leading to (exponentially) large query sizes and, thus, poor runtime performance.

In this chapter, we describe and study a new approach to data integration called Mapping Data to Queries or MDQ for short. The underlying principle of MDQ we call just-in-time data integration. The difference of just-in-time data integration to the traditional approaches is the point in time when the integration takes effect (i.e., when mapping rules are applied). In just-in-time data, mapping rules are applied at the latest possible point in time of query processing, at runtime of the query; while in the traditional approaches mapping rules are applied before query processing, at compile time or even earlier. At query runtime, just-in-time data integration has knowledge about the data, mapping rules, and the query itself, which allows to effectively optimize the integration. We call this approach just-in-time data integration because it carries out data integration at runtime in the same way as just-in-time compilation carries out optimizations at runtime. Just-in-time data integration is implemented by Mapping Data to Queries. MDQ interweaves query processing and data integration by matching mapping rules at query runtime and annotating the data with additional access paths. MDQ has better runtime performance than the traditional approaches (up to orders of magnitude) when
there are mixes of schemas.

In addition to performance, Mapping Data to Queries the following advantages:

1. The annotations introduced by MDQ can be cached and reused, allowing to move transparently between a virtual view and a materialized view of the integrated data. This may bridge the gap between federation techniques and data warehousing (i.e., transformations).

2. In addition to schema-level integration, MDQ can be adopted for data-level integration (e.g., entity resolution). This leads to single data integration engine that integrates data at the schema and data level holistically.

Both of these points are studied in the next chapter. This chapter, at first, focuses on the concepts and performance characteristics of just-in-time data integration and Mapping Data to Queries for schema-level integration.

In summary, we make the following contributions:

- We present the principle of just-in-time data integration, which carries out data integration at the runtime of a query.
- We present Mapping Data to Queries (MDQ), a new approach to data integration, which implements just-in-time data integration by indexing mapping rules and annotating data with additional access paths.
- We experimentally compare MDQ to both traditional approaches to data integration, query rewrite and transformations, using a synthetic benchmark and variations of the XMark [104] and TPoX [89] benchmarks.

The remainder of this chapter is structured as follows: Section 6.2 presents a motivating example that includes mixes of schemas. Section 6.3 explains assumptions we make to compare all data integration approaches. Sections 6.4 and 6.5 then describe design principles and details of just-in-time data integration and MDQ. Sections 6.6 and 6.7 discuss the traditional approaches based on data transformations and query rewrite respectively. Section 6.8 presents the results of performance experiments. Section 6.9 lists related work. Section 6.10 concludes this chapter.

6.2 Motivating Example

To illustrate a world of information with mixes of schemas, we use the following example from the health care domain. Figure 6.1 illustrates the example.
Figure 6.1: Patients with different health records register at the New York City hospital.

Figure 6.1, several patients wait in line at a hospital in New York City. The first patient in line, previously having been to hospitals in San Francisco and Zürich, registers her health record with the hospital system at the front desk of the hospital. The hospital system processes this health record immediately, forwarding a summary of the latest findings to the appointed doctor, generating reports for the financial department, and raising alerts in case of necessary immediate actions (such as quarantines, etc.)

The health record of the patient, shown in Figure 6.2, contains the history of findings from previous visits to hospitals in San Francisco and Zürich. The health record is represented in XML and consists of a mix of four different schemas. The first schema, identified by the \texttt{hl} namespace\footnote{More precisely, \texttt{hl} is a namespace prefix, which is associated with the namespace URI \texttt{http://hl7.org}.}, provides information about the patient itself. The second schema, identified by the \texttt{sf} namespace, describes the finding when the patient visited a hospital in San Francisco, including the date of the visit and the name of the appointed doctor. The third schema, identified by \texttt{zh}, describes the finding observed in Zürich. The fourth schema, identified by \texttt{tr}, represents medical treatments prescribed by doctors and may be a naming standard for treatments and medicine, and hence is used by hospitals in San Francisco and Zürich.

Via the use of mapping rules, the health record is translated into an integrated view (also shown in Figure 6.2), which is required by the system of the New York City hospital. Mapping rules define the translation from the source schemas of the different hospitals to the target schema of the New York City hospital. For example, the mapping rule \texttt{sf:finding} $\rightarrow$ \texttt{ny:finding} ($\texttt{sf:finding}$ “is a” \texttt{ny:finding}...
Figure 6.2: A health record of a patient is integrated via mapping rules and processed by a query. The health record consists of a mix of four schemas.
declare namespace ny="http://nyc.com";
declare namespace hl="http://hl7.org";
declare namespace sf="http://sf.com";
declare namespace zh="http://zh.ch";
declare namespace tr="http://tr.com";

hl:patient-record as $n → ny:record (attribute name $n/hl:name/text())
hl:medical-history → ny:history
sf:finding → ny:finding
sf:doc → ny:doc
sf:date as $d → ny:date (concat($d/sf:year, $d/sf:month, $d/sf:day))
zh:befund → ny:finding
zh:arztliste/zh:arzt → ny:doc
zh:datum as $d → ny:date (convert-date-from-zh($d))
tr:treatment → ny:treat

Figure 6.3: Mapping rules translating the health record of Figure 6.2 into the schema of the New York City hospital.

ing) defines that any sf:finding element in a source document translates to an ny:finding element in the target. All mapping rules needed to translate the health record in Figure 6.2 into the integrated view are shown in Figure 6.3. Syntax and semantics of the rules will be discussed in the next section. The integrated view itself may be materialized or exist only virtually depending on the data integration approach used.

Queries are processed using the integrated view of the health record. In our example in Figure 6.2, the query “Findings of the last three years” generates a summary of the latest findings. In XQuery this query would be implemented as follows:

```
declare namespace ny="http://nyc.com";
for $x in /ny:record/ny:history/ny:finding
  where xs:date($x/ny:date/text()) >
    current-date() - xs:yearMonthDuration('P3Y')
return $x
```

As can be seen, this query uses the New York City hospital’s schema (identified by the ny namespace), which is present in the integrated view. The result of the query may then be forwarded to the appointed doctor.
Figure 6.4: Overview of a system to integrate heterogeneous input data.

6.3 Assumptions

To compare existing approaches to data integration with Mapping Data to Queries, we make a number of assumptions concerning the system, data, and mapping rules.

6.3.1 System and Workload

The system we use to compare the different data integration approaches is illustrated in Figure 6.4. The system processes messages as input and sends messages that contain the query results as output. The system consists of a query processor that processes each input message. The query processor is supplied with queries. To process input messages in different schemas, the query processor is also supplied with mapping rules. As a result of query processing (and with the help of mapping rules), output messages are produced. How mapping rules are applied depends on the data integration approach used. It should be noted that this system supports all data integration approaches discussed in this chapter. In our running example, this system could be installed at the front desk of the hospital in Figure 6.1.

In particular, we make the following assumptions:

- The system knows all queries and mapping rules.
- Queries are expressed in XQuery 1.0 [12], mapping rules are described below.
- Messages are formatted in XML.
- Input messages can consist of mixes of schemas.
- The schemas of a message are not known until arrival at the system; schemas cannot be inferred by the data source or the message header.
• Each message is processed independently.
• Messages fit entirely into main memory.
• There are no indexes that index individual nodes of an XML document.

Figure 6.4 shows what is known and unknown to the system. Queries and mapping rules are known to the system before data processing begins. We assume that mapping rules have been created manually or using (semi-)automatic mapping tools. The system may therefore choose to rewrite queries using mapping rules, generate transformation programs, or index mapping rules for just-in-time data integration. Meanwhile, input messages and in particular the schemas of those messages are unknown to the system. Because messages may consist of mixes of schemas, schemas cannot be inferred by the data source or by the message header.

Messages can consist of a mix of a few schemas from any number of schemas. As pointed out in the introduction, there may be hundreds or thousands of different schemas describing the same or similar concepts. Messages can consist of a mix of few of these schemas, not necessarily all of them.

6.3.2 Mapping Rules

Mapping rules define how concepts of in source schema are mapped to concepts in the target schema. Because no standardized mapping rule language XML exists, we will shortly describe the syntax and semantics of the mapping rule language used in this work.

As mapping rules, we use rules of the following format:

\[ source-expr \text{ [as $var]} \rightarrow target-expr \text{ [content-expr]} \]

In this syntax, \text{as} and \text{→} are keywords. The source expression \text{source-expr} is any XPath 1.0 expression [29] with axis steps / (child::) and // (descendant-or-self::node()), and predicates only. The source expression can be specified as absolute path from the root of the document or as relative path from a location that becomes the context item. The target expression \text{target-expr} is a list of one or more node names (possibly with a namespace prefix) separated by /. Optionally, the target may contain an additional expression \text{content-expr}. \text{Content-expr} is an XQuery expression that returns a sequence of nodes. \text{Content-expr} may possibly involve element construction, joins, aggregation, user-defined functions, and so on. \text{Content-expr} may also involve free variables which are bound by the \text{as} clause in the source expression. This way, arbitrary XQuery expressions can be used to
create any target view. Last, \( \rightarrow \) may be replaced by \( \text{copy} \rightarrow \) to explicitly create a second copy of the source element that is being mapped.

Semantically, mapping rules define that all nodes accessible by \( \text{source-expr} \) are also accessible by \( \text{target-expr} \). More formally, evaluating \( \text{source-expr} \) starting at a context node \( c \in \text{doc} \) (\( \text{doc} \) being an XML document) yields a set of nodes \( S_c \). A mapping rule \( \text{source-expr} \rightarrow \text{target-expr} \) defines that all nodes \( s \in S_c \) will be returned when evaluating target expression \( \text{target-expr} \) starting at context node \( c \).

In case the target expression specifies an additional \( \text{content-expr} \), this expression is evaluated for each node \( s \in S_c \) and the resulting nodes are added to node \( s \) as child nodes. Elements and attributes will overwrite existing elements and attributes with the same name. Similarly, a text node will overwrite an existing text node.

A mapping rule \( \text{source-expr} \xrightarrow{\text{copy}} \text{target-expr} \) creates an additional copy \( s' \) of each node \( s \in S_c \) and each \( s' \) is returned when evaluating \( \text{target-expr} \).

Scoping of mapping rules is possible using namespaces and predicates within source expressions. If the source expression contains namespaces, mapping rules only apply to elements belonging to the respective namespace. Source expressions may also contain predicates that further limit the scope of mapping rules within a single namespace. For example, predicates may be used to horizontally partition contents of a source (see corresponding scenario in Table 6.1).

Although relatively simple, this mapping rule language supports all basic mapping scenarios defined by STBenchmark [4]. The basic mapping scenarios in STBenchmark stem from a careful analysis of integration needs of data exchange, data warehouses, XML publishing, schema evolution, and real-world mapping specifications. They capture common integration cases that occur in practice and have wide industry relevance [4]. Table 6.1 shows how to express these basic mapping scenarios using our mapping rule language. Table 6.1 only shows the scenarios that are covered in our experiments. The remaining mapping scenarios are shown in Table 6.7 in the end of this chapter (page 135).

Mapping rules have a number of restrictions. (1) Only elements can be mapped. (2) Source expressions of mapping rules consist of a subset of XPath 1.0 with axis steps / and //, and predicates only. Other source expressions (e.g., the parent of an element) cannot be specified. (3) Mapping rules do not allow free variables in the target expression that are not bound by the \( \text{as} \) clause in the source expression. Hence, our rules are functions producing a single target data set (not a set of incomplete data or possible worlds as is common with mapping languages such as source-to-target tuple-generating dependencies [41, 75]). We restrict our approach to such rules so that the target can be queried with standard query languages (in our case XQuery) without the use of certain answer semantics. (4) Mapping rules are not ordered and may be executed in any order.
Table 6.1: Expressing the basic mapping scenarios of the STBenchmark [4] using mapping rules.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Source and target schema</th>
<th>Mapping rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliasing</td>
<td>s:protein*</td>
<td>t:protein*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td>Constant value generation b</td>
<td>s:source</td>
<td>t:target</td>
</tr>
<tr>
<td></td>
<td>“July 4”</td>
<td>t:date</td>
</tr>
<tr>
<td>Horizontal partitioning c</td>
<td>s:gene*</td>
<td>t:gene*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td></td>
<td>s:gene</td>
<td>t:synonym*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td>Vertical partitioning d</td>
<td>s:reaction*</td>
<td>t:reaction*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td></td>
<td>s:definition</td>
<td>t:cofactor</td>
</tr>
<tr>
<td></td>
<td>t:information*</td>
<td>t:cofactor</td>
</tr>
<tr>
<td>Unnesting e</td>
<td>s:reference*</td>
<td>t:publication*</td>
</tr>
<tr>
<td></td>
<td>s:title</td>
<td>t:title</td>
</tr>
<tr>
<td></td>
<td>s:author*</td>
<td>t:name*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td>Nesting f</td>
<td>s:publication*</td>
<td>t:reference*</td>
</tr>
<tr>
<td></td>
<td>s:title</td>
<td>t:title</td>
</tr>
<tr>
<td></td>
<td>s:name*</td>
<td>t:author*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:name</td>
</tr>
<tr>
<td>Atomic value manipulation g</td>
<td>s:contact*</td>
<td>t:contact*</td>
</tr>
<tr>
<td></td>
<td>s:name</td>
<td>t:firstname</td>
</tr>
<tr>
<td></td>
<td>s:address</td>
<td>t:lastname</td>
</tr>
<tr>
<td></td>
<td>s:street</td>
<td>t:address</td>
</tr>
<tr>
<td></td>
<td>s:city</td>
<td>t:address</td>
</tr>
</tbody>
</table>

\[\text{An instance of a source is renamed (or aliased) in the target.}\]
\[\text{Contents of a source are partitioned into two or more fragments in the target.}\]
\[\text{Records of a source are vertically split into two or more records in the target.}\]
\[\text{Records of a source are unnested (or flattened) in the target.}\]
\[\text{Records of a source are nested in the target.}\]
\[\text{Atomic values of a source are split or combined into one or more atomic values in the target.}\]
We claim no novelty with regard to the mapping rules. There are many integration frameworks and ways to define data integration logic (e.g., Clio [40], Clip [99], HepToX [15], Piazza [55], Orchestra [63], and numerous logical foundations including Calvanese et al. [24]). We have merely adopted best practices here and applied them to the syntactic needs of XML data and XQuery expressions. Most importantly, we were able to implement these kinds of mapping rules in all three data integration approaches studied in this chapter.

6.3.3 Conflicts in Mapping Rules

Mapping rules may have conflicts. It is the responsibility of the mapping rules developer to avoid these.

Mapping rules may map the same source expression multiple times. As mapping rules are not ordered, it is undefined in which order mappings are executed. For example, in the vertical partitioning scenario in Table 6.1, mapping rules \texttt{s:reaction} \rightarrow \texttt{t:reaction} and \texttt{s:reaction} \rightarrow \texttt{t:information} contain the same source expression and map the same element twice. Thus, it is unclear whether \texttt{s:reaction} maps to \texttt{t:reaction} or \texttt{t:information}. As a solution, the mapping rule developer explicitly modifies the second mapping rule to \texttt{s:reaction} \rightarrow \texttt{t:information} to create a second copy of \texttt{s:reaction}, which then maps to \texttt{t:information}.

Mapping rules may contain cycles. There may be cycles within one mapping rule, for example in the rule \texttt{a} \rightarrow \texttt{a/a}, or across multiple mapping rules, for example if one rule adds a new node (using \texttt{content-expr}) that is then mapped to another name by a second rule for which the first rule applies again (adding another new node and so on). Such rules will lead to infinite loops during mapping rule/query execution. It is the responsibility of the mapping rule developer to avoid cycles in mapping rules; similar to how it is the responsibility of any programmer to avoid infinite loops in program code.

6.4 Just-in-time Data Integration

The key idea explored in this chapter is to carry out data integration when a message is first touched as part of processing a query—just in time. We call this principle just-in-time data integration. During query processing, both the data and the query are known, and unnecessary integration work can be avoided: (1) Data that is not relevant for a query need not be transformed. In this respect, just-in-time data integration outperforms the transformation approach. (2) Mapping rules
that never become applicable do not slow down query processing. In this respect, just-in-time data integration outperforms the query rewrite approach. Finally, as we will show, it is possible to cache results of just-in-time data integration. In this way, a particular data integration effort needs to be carried out only once even if several queries process the same message. Again, this is an advantage of just-in-time data integration as compared to the query rewrite approach.

6.4.1 Implementation

The technique to implement just-in-time data integration is called Mapping Data to Queries. MDQ interweaves data integration and query processing based on two important ideas. First, MDQ indexes the source expressions of mapping rules. This way, mapping rules can be matched efficiently at query runtime. Second, MDQ annotates the data with additional access paths, rather than transforming the data. The annotations are a light-weight way to represent the effects of applying mapping rules to the data. This optimization becomes possible because we are applying MDQ to a semi-structured (i.e., tree) data model. Caching these annotations ensures that the data integration effort needs only to be carried out at most once for each data item. The main working principle of MDQ is explained in detail in the next section.

6.4.2 Discussion

The advantage of just-in-time data integration and its implementation, MDQ, is that it scales well with the number of schemas. MDQ is particularly effective if documents contain mixes of schemas. In extreme cases, when there are documents with mixes of schemas and thousands or tens of thousands of mapping rules, MDQ outperforms traditional data integration approaches by orders of magnitude. Additionally, MDQ performs better than the query rewrite approach if the queries are complex and involve many path expressions or joins. Furthermore, using MDQ, queries need not be recompiled if new mapping rules are added or existing mapping rules are modified.

The main disadvantage of MDQ is that it involves modifications to the core of a query processor. All key components including the storage manager, the query compiler, and the runtime system need to be adjusted. The next section describes the adjustments we make to implement MDQ.
6.5 Mapping Data to Queries

This section describes how an off-the-shelf query processor can be extended for just-in-time data integration with MDQ. Since we use XQuery as a query language, we chose to extend the Saxon XQuery processor [103]. Saxon is the most popular open-source XQuery processor available today. Saxon follows a textbook architecture for an XQuery processor [44] so that we expect that other XQuery processors can be extended in the same way. The only assumption is that the scan operator of the XQuery processor recursively scans the labeled tree representation of an XML document in depth-first left-to-right order. This section describes the extensions to the Saxon storage manager (needed to implement annotations), the Saxon runtime system (needed to interweave query processing and data integration), and the way we index mapping rules (needed to match mapping rules at query runtime and ultimately scale with the number of schemas).

6.5.1 Storage Manager

To implement the MDQ approach, the storage manager, which maintains the internal representation of an XML document, must be extended to store annotations generated by the evaluation of mapping rules on the data.

Figure 6.5 illustrates how MDQ annotates data with the help of an example. Figure 6.5(a) shows the XML snippet of the San Francisco finding of Figure 6.2 represented as a labeled tree. The labels of edges represent the names of attributes or elements in the original XML document. Leaf nodes contain values (e.g., the string “Dr. Peterson”) of the original XML snippet. Labeling edges instead of nodes allows to annotate the tree with additional edges between existing nodes (i.e., new access paths). Figure 6.5(b) shows the annotated tree after applying the mapping rules in Figure 6.3. For example, the rule \(\text{sf:finding} \rightarrow \text{ny:finding}\) adds an additional edge to the tree to represent that the sf:finding node can also be accessed by a path over \(\text{ny:finding}\). Similarly, rules \(\text{sf:doc} \rightarrow \text{ny:doc}\) and \(\text{tr:treatment} \rightarrow \text{ny:treat}\) each add an additional edge to the tree. The mapping rule \(\text{sf:date as$ d \rightarrow ny:date \ (\text{concat($d/sf:year, \$d/sf:month, \$d/sf:day))}\) not only adds an edge \(\text{ny:date}\) to the tree but also adds an additional text node “2011-01-04”, which contains the result of the concat function.

Figure 6.6 illustrates the four ways a mapping rule can impact a labeled tree. The simplest case is shown in Figure 6.6(a) for aliasing rules of the form \(a \rightarrow x\), which map one name to another name. In this case, an additional edge \(x\) from the parent to the child is added (just as for \(\text{sf:finding} \rightarrow \text{ny:finding}\) in Figure 6.5(b)). If the source expression of the rule involves a path expression (e.g., \(a/b\)), then the
edge can go across several levels of the tree; this pattern is shown in Figure 6.6(b). Since the source expression of a rule only involves forward axes / or //, an edge always goes from a node to one of its descendants (never to an ancestor)—so the annotated tree is always acyclic. Figures 6.6(c) and 6.6(d) show how new nodes can be created through the application of a mapping rule. Nodes are generated if the target expression of a rule involves a path expression (e.g., x/y in Figure 6.6(c)) or the optional content-expr (e.g., (<y/>) in Figure 6.6(d), which creates a new element y).
Figure 6.6: Expansion cases.

Figure 6.7: Copy rule.

Figure 6.7 shows how the mapping rule $a \rightarrow x$ impacts an XML tree. Instead of adding an additional edge from the parent of $a$ to the child, this rule adds a new node and copies all possible content. The new node may contain additional child nodes using expression content-expr (as, e.g., in the vertical partition scenario in Table 6.1.)

Figure 6.8 illustrates that a node can have several annotations. For instance, node $a$ of the annotated tree of Figure 6.6(a) could have three incoming edges if there were an additional rule $x \rightarrow y$: (1) the $a$ edge from the original XML snippet, (2) the $x$ annotation generated from the $a \rightarrow x$ rule, and (3) a $y$ annotation generated from the $x \rightarrow y$ rule. In this example, the $x$ annotation is generated before the $y$ annotation; only the application of the $x$ rule makes the $y$ rule applicable.

The annotations in the annotated tree are not ordered. Annotations are only needed during query processing to detect whether a particular node of an annotated tree matches a query. For this purpose, at least one annotation needs to match. If several annotations match, the result is the same as if only one annotation matches (because of duplicate elimination as explained below) and it is irrelevant which annotation was detected first to match the query. As a result, the annotations do not need to be ordered.

An annotated tree can be serialized to XML (or JSON or any other serialization format) in exactly the same way as a classic (un-annotated) tree in order to produce query results. As part of serialization, the annotations are simply ignored. As a result, in the example of Figure 6.5(b), the query //ny:finding returns a sf:finding element with sf:date, sf:doc, and tr:treatment sub-elements. The result does not contain any ny:finding or ny:date elements. Our implementation allows...
Figure 6.8: Multiple annotations.

that behavior to be switched to ignore original edges and only serialize annotations or edges conforming to a certain namespace. With this modified behavior, serializing the annotated tree in Figure 6.5(b) yields the integrated view shown in Figure 6.2.

For practical purposes, there are many alternative ways to implement labeled trees in XQuery processors. Saxon, for instance, uses the TinyTree model [66]. Annotated trees, as required for MDQ, are no longer trees in the strict sense; they are acyclic directed graphs. Nevertheless, all the implementations that we are aware of can easily be extended to implement such directed acyclic graphs with annotations. In our implementation for Saxon, we extended the TinyTree model such that multiple edges can point to the same node.

6.5.2 Query Compiler

Although the change in the storage manager from a tree to an acyclic graph with annotations seems significant, only two adjustments must be made to the query compiler of an XQuery processor. The two changes involve additional duplicate elimination and sorting of a query result in document order.

Additional duplicate elimination becomes necessary when a query involves union expressions. For instance, duplicate elimination becomes necessary for the expression `//(sf:finding | ny:finding)` (i.e., the union of sf:finding and ny:finding) when applied to the annotated tree of Figure 6.5(b). Without duplicate elimination, the node with node id #1 would be returned twice as a query result because this node matches both paths. With duplicate elimination, node #1 is correctly returned only once. Therefore, whenever a query contains a union expression, the query compiler adds an additional duplicate elimination operator to the compiled query plan.

Sorting in document order is necessary if queries contain recursive operators and the document contains nested elements with the same name. A recursive operator (recursively) visits the nodes of an XML tree at different levels. For example, the descendant-or-self operator recursively visits all children, and the children of the children, and so on. In an annotated tree, the result of a recursive operator is not
in document order if there are nested elements that are annotated with edges of the same name. For example, given the document `<a><a/></a>` and the mapping rule `a → x` (illustrated in Figure 6.9), the recursive operator of a query `//x` outputs node #2 before node #1. Sorting the result of this query in document order is necessary. Therefore, whenever a query contains a recursive operator, the query compiler adds an additional document order operator to the compiled query plan.

Obviously, duplicate elimination and sorting in document order are not always needed. Additional duplicate elimination is not necessary for queries without union expressions (e.g., `//ny:finding`). Sorting can be avoided if there are no recursive operators in a query.

### 6.5.3 Runtime System

Mapping Data to Queries works at the runtime of a query. Figure 6.10 shows how the runtime system of a traditional query processor is extended to support MDQ. The runtime system of a traditional query processor (Figure 6.10(a)) is composed of a query plan interpreter and a storage manager. The query plan interpreter carries out a query plan by executing each operator of a query (e.g., scans and joins). Thereby, it fetches nodes from the storage manager. The storage manager maintains the labeled tree representation of the XML data. This architecture is extended in three ways (Figure 6.10(b)):

1. The storage manager is extended for annotated trees, and keeps a flag for each edge in the annotated tree if it has been traversed by any scan operator.

2. A rule manager is added, which maintains an index of mapping rules, and creates new edges and nodes in the annotated tree for any matching mapping rules.

3. The query plan interpreter is extended to notify the rule manager so that the interpretation of mapping rules in the rule manager and the interpretation of queries is interweaved.
The remainder of this subsection explains in more detail how the interpreter and rule manager interact during query processing using MDQ.

A naive way to implement MDQ is to apply all rules to an XML tree eagerly, before processing queries on that tree for the first time. That is, when the tree representation of an XML document is built, immediately all rules are applied creating a completely annotated tree. After that, all queries can be interpreted using the annotated tree in the same way as in a traditional query processor.

This eager MDQ approach is similar to the transformation approach, which also carries out data integration eagerly before query processing; the only difference is that more light-weight annotations, rather than a whole new copy of the tree, would be created as part of an eager MDQ implementation. Correspondingly, the eager MDQ approach has roughly the same advantages and disadvantages as the transformation approach. On the positive side, this eager MDQ approach is simple to implement (just like the transformation approach). On the negative side, it can exhibit poor performance because it may involve wasted work (just like the transformation approach, see Section 6.6).

The alternative to the eager approach is to interweave the processing of mapping rules with the processing of queries. Annotations are added just-in-time. Just-in-time data integration requires extensions to the runtime system as shown in Figure 6.10(b). Specifically, the scan operator of the query plan interpreter must be modified. The scan operator traverses the XML tree by fetching nodes from
the storage manager in depth-first left-to-right order. It is extended to notify the rule manager of each edge that is traversed. All the other operators (e.g., joins and sorting) do not need to be modified.

Scanning an XML tree starts at the root and involves iterating through parent-child relationships of each visited node. Whenever the scan operator traverses an edge for the first time, it notifies the rule manager and informs it of the traversed edge. Via a flag on each edge, the scan operator remembers which edges it has already traversed. Each edge is reported to the rule manager only once. The rule manager, in turn, checks which mapping rules apply for the traversed edge (and the node that the edge points to). If a mapping rule applies, the rule manager expands the annotated tree (i.e., adds edges and nodes as defined by the mapping rule). After the notify call to the rule manager returns, the call to the scan operator will also return. New edges, which were created as part of the notify call to the rule manager, are visited in later iterations of the scan operator.

6.5.4 Rule Manager and Indexes

Whenever the rule manager gets notified of an edge that is traversed by the scan operator, the rule manager checks if for this edge any mapping rules match. To do so, the rule manager does the following:

1. It keeps an index of mapping rules using YFilter.
2. It queries the index to check if the reported edge and any edges in a (bounded) subtree beneath the reported edge match any mapping rules.

The rule manager indexes mapping rules using YFilter [38] (see also Section 2.5). A single non-deterministic finite automaton (NFA) is constructed to index the source expressions of all mapping rules. The accepting states of the NFA are annotated with the target expressions of the corresponding mapping rules and a pointer to the state of the NFA that represents the context node. The context node is needed to detect the starting point of the new edge that is added to the annotated tree. When querying YFilter, the NFA transitions to new states. Whenever the NFA reaches an accepting state, it indicates that the corresponding mapping rule matches. The mapping rule’s target expression is evaluated to compute the new edges and possibly new nodes. Edges and nodes are added to the annotated tree. Figure 6.11 shows the NFA for the single mapping rule sf:finding \(\rightarrow\) ny:finding. Whenever it is queried for a sf:finding edge, the NFA reaches accepting state \(q_2\) and the mapping rule is applied to the annotated tree. That means, the rule
manager adds a ny:finding edge to the tree. The new edge goes from the context node (stored in state \( q_1 \)) to the target node of the sf:finding edge.

Figure 6.12 shows the NFA of Figure 6.11, which is extended with two more mapping rules, \( sf: \text{date} \rightarrow ny: \text{date} \) and \( zh: \text{arztliste} / zh: \text{arzt} \rightarrow ny: \text{doc} \). As in the automaton of Figure 6.11, the NFA of Figure 6.12 reaches accepting state \( q_2 \) whenever a sf:finding edge is queried; in this case, the first mapping rules is applied and operates in the same way as in the NFA of Figure 6.11. In addition, the NFA of Figure 6.12 reaches accepting state \( q_3 \) whenever a sf:date edge is queried; in this case, the second mapping rule is applied. Finally, the NFA reaches accepting state \( q_3 \) if a sequence of zh:arztliste and zh:arzt edges are queried; in this case, the third mapping rule is applied. If two or more rules share the same accepting state, these rules can be executed in any order because annotations in the annotated tree are unordered as explained in Section 6.5.1.

The rule manager queries the mapping rule index for the edge reported by the scan operator. Furthermore, the rule manager traverses a subtree beneath the reported edge to check if any other mapping rules match that have a source expression with multiple steps. The subtree is bounded in height. The height of the subtree equals the length of the longest source expression of any mapping rule (i.e., the source expression with the most steps). After the rule manager checked the reported edge and the bounded subtree beneath the reported edge, all mapping rules which involve this edge have been found and any changes have been applied to the labeled tree. Therefore, every edge in the tree needs to be reported to the rule manager only once.

This strategy of matching mapping rules (i.e., additionally checking bounded subtrees of reported edges) is general enough to handle any type of XQuery queries that are evaluated by the query plan interpreter. In particular, nested for loop expressions and any variable bindings in XQuery, which result in “jumping” back and forth in an XML tree during query evaluation, are handled properly by the rule manager.

However, Mapping Data to Queries requires that an XML document is scanned starting from the root. Our approach is, thus, not applicable if intermediate (non-root) nodes of XML document are indexed. Saxon does not support such fine-
grained indexes, nor do most other XQuery processors we are aware of. Most XQuery processors index on the granularity of whole XML trees. Adjusting the MDQ for indexes that reference intermediate nodes of a tree is left for future work.

In summary, MDQ implements just-in-time data integration by applying mapping rules at the runtime of a query. To do so, the source expressions of mapping rules are indexed to determine if a mapping rule matches an XML document, which is currently processed. In case a mapping rule matches, MDQ will annotate the labeled tree representation of the XML document with new edges. These edges are scanned by the query plan interpreter just as any other edges. Ultimately, all necessary mapping rules will be applied, and the query will be answered.

In the following, we discuss the two prominent traditional approaches to data integration, transformations and query rewrite.

### 6.6 Transformations

In data integration, one traditional approach is to transform diverse input messages into new messages that conform to the target schema. The target schema is used by application developers to write queries against. In our motivating example, the target schema is the schema of the New York City hospital. Mapping rules define how to map elements and attributes of input messages to the target schema. According to these mapping rules, each incoming message is transformed into a new message, which complies to the target schema. After the transformation, the new message can be queried using normal query processing/database techniques.
6.6.1 Implementation

The transformation approach is implemented using transformation programs. Transformation programs transform input messages into new messages in the target schema. Thereby, transformation programs apply the mapping rules to individual elements to create new elements in the target schema. Figure 6.13 illustrates a transformation program that transforms a medical history such as the one in Figure 6.2 into new history in the schema of the New York City hospital. The transformation program in Figure 6.13 is written in XQuery. Specifically, this program does the following: First it constructs a new ny:history element in the beginning of the transform-history function. Then, for each finding in the medical history, it uses a big if-then-else expression that (1) simply copies all ny:finding elements into the new element (the then branch); (2) transforms sf:finding elements into ny:finding elements (the first else branch); and (3) transforms zh:befund elements into ny:finding elements (the second else branch). More else branches are needed to transform findings in other schemas. The transformation program is derived from the mapping rules in Figure 6.3.

Transformation programs become more complex for more schemas. There will be more branches in the if-then-else expression for more schemas. There will be nested if-then-else expressions if elements contain child elements in different schemas. There might even be additional calls to other transformation functions, which themselves can have (nested) if-then-else expressions. In Figure 6.13, for example, there are additional calls to a transform-treat function, which transforms all variations of treatment elements.

6.6.2 Discussion

The advantages of the transformation approach are reuse of transformed messages and scalability in terms of data size. Once a message has been transformed, the transformed message can be reused by many queries without transforming the message again. The transformation program has to be run only once for each message. Furthermore, transformation programs are typically tailored to transform huge amounts of data (usually over night) and store the integrated data in a database or a data warehouse for convenient ad-hoc query processing thereafter. Thus, the transformation approach is typically used in ETL workflows (extract-transform-load) to load data into a data warehouse.

The disadvantage of the transformation approach is the high processing overhead of the transformation. Transforming messages is time consuming because each element of an input message must be analyzed, and for each element a new element
Figure 6.13: A program to transform a medical history into the target schema.

must be created in the new message. This process cannot be optimized with respect to a subsequent query. Each input message is completely and eagerly transformed into a new message before it is queried. The transformation approach eagerly transforms the complete message independent of whether all parts of the message are needed to produce the query result. Therefore, the high price of transforming messages must be paid for every time.
6.7 Query Rewrite

The other traditional approach is to rewrite the queries to match all possible schemas (and mixes of schemas) of input messages. In this approach, the data is not transformed or modified. Queries are rewritten at compile time—before they are executed—based on the mapping rules. The key idea is to generate union expressions whenever the right-hand side of a mapping rule is subsumed by a subexpression of the query. A simple example is the query //ny:finding, which is rewritten to //((ny:finding | sf:finding) given the rule sf:finding → ny:finding. The operator | is the union operator in XQuery. Thus, the query matches findings in both schemas.

Rewritten queries can become very large. Continuing the example, given mapping rules sf:finding → ny:finding and tr:treatment → ny:treat, the query //ny:finding/ny:treat is rewritten to the following query:

```
```

Because we assume mixes of schemas, the rewritten query must contain each possible mix of findings and treatments. The subexpression sf:finding/ny:treat could be removed if we were certain that the ny:treat element is never a child of sf:finding. However, because schemas of messages are not known in advance, such optimizations are not possible (in the scenarios considered in this thesis).

6.7.1 Implementation

One way to implement the query rewrite approach is based on Miklau and Suciu [83]. The implementation consists of two steps: (1) generating union expressions for subexpressions of a query based on mapping rules, and (2) substituting the generated union expressions for the subexpressions in the query. For every subexpression of a query that subsumes the right-hand side of at least one mapping rule (i.e., the target expression), a union expression must be generated which involves the original subexpression and an expression based on the left-hand side of the mapping rule (i.e., the source expression). If a subexpression subsumes the target expression of several mapping rules, then the union expression naturally involves the source expressions of all matching mapping rules.

Table 6.2 lists the templates we need to generate union expressions for our experiments in Section 6.8. The first three templates are straightforward: for simple
Table 6.2: Union expression templates (for a query that involves an \( x \); e.g., \( //x \)).

<table>
<thead>
<tr>
<th>Mapping rule</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \rightarrow x )</td>
<td>((x</td>
</tr>
<tr>
<td>( a[b] \rightarrow x )</td>
<td>((x</td>
</tr>
<tr>
<td>( a/b \rightarrow x )</td>
<td>((x</td>
</tr>
<tr>
<td>( a \text{ as } $a \rightarrow x (\ldots))</td>
<td>((x</td>
</tr>
</tbody>
</table>

path expressions, the union expression is constructed using the source expression of the mapping rule. If the target expression of the mapping involves additional element construction (the last template of Table 6.2), then an alternative in the union must be generated for each instantiation of the variable bound to the source expression of the mapping rule. This can be implemented in XQuery with a simple FLWR expression.

The second step, after the generation of union expressions, is to substitute steps in the path expressions of the original query with the generated union expressions. Every step expression of the original query is replaced with the corresponding union expression (which is found using a path containment checking algorithm [83]).

Table 6.2 does not deal with the case that the target expression of a mapping rule is a multi-step path expression; for example, \( a \rightarrow x/y \). A query involving \( x/y \) would be rewritten to \((x/y | a)\). This case is special because checking path containment for XQuery is difficult in the general case [83]. In our experiments, we manually rewrote queries to support these kind of mapping rules.

6.7.2 Discussion

Query rewrite shows good performance when applied in a relatively static setting where schemas of data sources are known in advance. That is, when querying data from well-known data sources. A query would then be rewritten for each particular data source only (without involving union expressions), optimized given characteristics of the query and the data source, and results would be combined [52]. Typically, in such a static setting, one would apply query rewrite. However, in this work, data characteristics are much more flexible (i.e., mixes of schemas) and schemas of messages are not known in advance.

The main disadvantage of query rewrite (in the scenarios in this work) is the large size of rewritten queries and therefore poor runtime performance of rewritten queries. The more schemas of input messages that need to be considered, the larger individual union expressions become, and thus the larger the rewritten queries.
Furthermore, because we assume that input messages may consist of mixes of schemas, rewritten queries must consider all possible combinations of schemas (as in the example above), which further leads to larger rewritten queries. In our running example, if every hospital and insurance company defines its own schema, then the size of even simple queries grows at least linearly with the number of hospitals and insurance companies. Large sizes of rewritten queries lead to poor runtime behavior as we will show in our experiments.

Because query rewrite takes effect at compile time, before the data is known, it is not possible to optimize the rewritten query based on the input message. In particular, given the assumption that the schema(s) of an input message is not known in advance, it is not possible create a rewritten query which takes into account only the schemas that the input message actually consists of.

However, after the rewrite, the query compiler can simplify and optimize the query. After query rewrite, all state-of-the-art query processing techniques are applicable. In fact, the query rewrite approach can be implemented using a pre-processor and an off-the-shelf database management system or stream processing engine of choice.

### 6.8 Experiments and Results

We implemented all three data integration approaches (MDQ, transformations, and query rewrite) using Saxon, a main memory XQuery processor [103]. This section presents the results of performance experiments using a synthetic benchmark derived from the health care domain, and variants of the XMark benchmark [104] and the TPoX benchmark [89], which include mixes of schemas.

#### 6.8.1 Software, Hardware, and Metrics

**Software**

All approaches were implemented in Java 1.6 on top of Saxon 9. Saxon is an XQuery processor written in Java. It is mainly used in a message processing scenario in which XML messages are processed one at a time using a set of XQuery expressions. This resembles a streaming use case, where messages are analyzed by (continuous) queries. Saxon also allows the loading of data into its in-memory store. Once the in-memory store of Saxon, the data can be queried just as in any other database scenario. In our performance experiments, we studied both scenarios: (1) streaming data processing of continuous queries and (2) traditional (pull-based) database processing (however, without indexing).
To implement just-in-time data integration with MDQ, we extended the Saxon compiler, storage manager, and query plan interpreter as described in Section 6.5. Without any mapping rules the original Saxon and the MDQ-enabled Saxon show roughly the same performance, so our implementation provides a fair comparison of MDQ, query rewrite, and transformations in Saxon. We expect that similar extensions to other XQuery processors would produce similar results because we did not exploit any features specific to Saxon in our implementation.

The query rewrite approach was implemented using a benchmark-specific preprocessor to generate the rewritten queries. That is, the pre-processor was designed specifically for the synthetic, XMark, and TPoX benchmarks and the corresponding mapping rules used in our performance experiments. We made sure in an additional manual step that the best possible query rewritings were used in all cases. In particular, we used union expressions of the form \(\text{//}(a \mid b)\) instead of \(\text{//}a \mid \text{//}b\) to involve the costly descendant-or-self operator \(\text{//}\) only once. The rewritten queries were then executed using Saxon as is; that is, Saxon was not changed.

The transformation approach was implemented using XQuery transformation programs such as the one in Figure 6.13. The transformation program was executed, again, using Saxon. The complete input message was transformed before executing the actual query. The result of the transformation was kept entirely in main memory without materializing intermediate results to disk to get the best possible performance. In our synthetic health care benchmark, we used the transformation program shown in Figure 6.13, which transforms individual findings using a big if-then-else statement. In the XMark and TPoX benchmarks, we used the following, more generic transformation program:

```xquery
declare function transform($elem) {
    if (name($elem) = 's1:person')
        then
            <ta:person>{
                for $child in $elem/*
                    return transform($child)
            }</ta:person>
        else if (name($elem) = 's2:person')
            then
                ...
        }
```

This transformation program compares the name of the element $elem to the possible element names of foreign schemas (s1:person and s2:person in this example), transforms the element to the target schema (ta:person), and recursively transforms all its children elements (via recursive calls to `transform`).
As a baseline, an ideal world was studied in all experiments. In such an ideal world there is no heterogeneity. All data and queries conform to the same schema and no mapping rules are needed. These experiments were carried out with “off the shelf” Saxon. Obviously, the performance in an ideal world is better than the performance that can be achieved using any of the three data integration approaches.

Hardware

All experiments were carried out using a single machine with an AMD 2.4 GHz CPU and 6 GB RAM running RedHat Enterprise 4.

Metrics

All experiments study runtime performance. We measure message throughput (in messages per second) for the streaming scenario and query response time (in milliseconds) for the traditional database scenario.

The measured performance metrics contain:

- Annotating messages during query runtime (for MDQ)
- Transforming incoming messages (for the transformation approach)

Incoming messages did not contain any annotations for the MDQ approach. That is, the effort to create these annotations was included as part of the runtime performance of MDQ. If several queries were executed on a single message, annotations were cached between queries so that the same annotation did not have to be generated twice. The cost for transforming messages was included as part of the runtime performance of the transformation approach. If several queries were executed on a single message, the message was transformed only once for all queries.

The measured performance metrics do not contain:

- Query compilation time
- NFA creation time
- XML message parsing time
In all experiments, the queries were pre-compiled. Furthermore, the NFA that indexes all mapping rules was already created. XML messages were sent in a binary format (i.e., pre-parsed). That is, query compilation, NFA creation, and XML message parsing were done before query processing and were not measured as part of the throughput and response times.

6.8.2 Benchmark Environment

Our experiments were performed using three different benchmarks: A synthetic benchmark inspired by the health care domain, the XMark benchmark [104], and the TPoX benchmark [89]. The synthetic benchmark and XMark were used to study data integration for continuous query processing on data streams. The TPoX benchmark was used to study more traditional database scenarios with ad-hoc queries.

The synthetic benchmark models a scenario of health records processed by the front desk of a single hospital (as in our example in Figure 6.1). Each health record contains a mix of different findings generated by the schema generator of the benchmark, and the number of schemas of the findings is varied in the performance experiments. Each health record contains a header and a body. The header contains information about the name, address, and insurance of a patient. The structure of the header is identical for all records. The body of a health record contains the patient history: a set of findings in different schemas. To simulate heterogeneity in this benchmark, the schema generator outputs findings in different schemas. To map one finding of a different schema to the target schema, five mapping rules are needed. The mapping rules used were aliasing rules (in 65% of all cases), nesting and unnesting (25%), and manipulating atomic values (10%). The average size of a health record (serialized as XML) was 11 KB.

As continuous queries, the synthetic benchmark used the seven queries defined below. These queries vary in complexity (i.e., involvement of joins, sorting, and element construction). Queries 1 to 4 are simple queries using XPath expressions only. Queries 5 to 7, in turn, are complex queries that involve nested loop joins, sorting, and expensive constructions of new elements. The queries are written in XQuery using the target schema. The queries are defined in Table 6.3.

The XMark benchmark models an online auction system, and messages involve bids on items [104]. The queries in XMark vary from simple path expressions that select certain properties of a message to complex queries that involve joins and aggregation. We implemented all twenty queries of the XMark benchmark and studied them individually (i.e., only one query is applied to each message) and as a whole (i.e., all twenty XMark queries are applied to each messages). If not stated
otherwise, the XMark data generator was set to generate messages of size of about 1.1 MB. In order to study the effects of different message sizes, this parameter was varied from a few KBs to 100 MB in a separate experiment.

The TPoX benchmark models a finance institution and involves a database of financial orders [89]. The queries in TPoX vary from simple selection of orders (based on their order number) to more complex queries (e.g., aggregates on a subset of orders). Due to the nature of the queries and the TPoX data generator, the TPoX benchmark cannot be applied as a measure of the performance of data stream management systems. As a result, we used this benchmark in our experiments to study the trade-offs of the alternative data integration approaches in a more traditional database scenario. That is, we loaded the TPoX database into the Saxon in-memory store, ran the TPoX queries using Saxon, and measured the response time of each TPoX query for all three approaches. Since Saxon has no support for indexes and its store is main-memory-based, we used the smallest possible TPoX database, which contains 200 MB of XML data.

To simulate heterogeneity in the XMark and TPoX benchmarks, we took the original XMark and TPoX schemas as a starting point and created new schemas by renaming all the element and attribute names. We create mapping rules to map the renamed elements and attributes back to the names in the original schema. For the XMark benchmark, 46 mapping rules are generated for each new schema. For the TPoX benchmark, each new schema involves 10 mapping rules. Depending on the query, however, only a fraction (between 5–10 for XMark, 2–10 for TPoX) of these mapping rules were applicable for each query. Messages contain schema mixes in such an extreme way that it is not possible to optimize the transformation program to parts of the message (as done when transforming findings in
our synthetic benchmark). Therefore, we use the generic recursive transformation program explained above.

The XMark and TPoX benchmarks were used to study the performance trade-offs of a variety of queries, as well as database and message configurations (i.e., size, data skew, and formats). Due to the complexity of the queries, however, it was not possible to study different types of mapping rules. Therefore, in these benchmarks we only used aliasing rules (i.e., $a \rightarrow x$).

In all experiments reported in this section, we varied the number of schemas to study how well the alternative data integration approaches scale in this dimension. (Obviously, query performance in an ideal world is independent of this parameter, since there is only one schema.) From discussions with organizations that use data integration technology, it seems that tens to hundreds of schemas are common today. With emerging standards such as HL7 and XBRL, we expect that data integration of as much as thousands of schemas will be required in the near future. As a result, we varied the number of schemas from tens to thousands, to cover the broad range of present and near-future scenarios.

### 6.8.3 Synthetic Benchmark

**Simple Query**

Figure 6.14 shows the results of our synthetic benchmark. Figure 6.14(a) shows the throughput of the three alternative approaches if only one simple continuous query (Query 1) is processed with a varying number of schemas. The results look very similar for the other simple queries (i.e., queries 2, 3, and 4). As a baseline, Figure 6.14(a) also shows the message throughput for an ideal world of no heterogeneity. Figure 6.14(a) shows a number of effects. Overall, MDQ shows the best performance, but the differences between query rewrite and MDQ are marginal because the penalty of rewriting simple queries is not very large; that is, the size of the rewritten query does not explode. While the performance of query rewrite degrades with the number of schemas, the performance of MDQ stabilizes at $\approx 1200$ messages per second. The performance of the data transformation approach is fairly independent of the number of schemas because the transformation program is tailored to transform individual findings without recursive transformation calls. At the same time, the transformation approach is clearly outperformed by the other approaches in this experiment because the overhead to transform every incoming message is high compared to the effort to actually process the message with a simple query.

For a single schema, the transformation approach performs worse than the ideal
Figure 6.14: Scalability with the number of schemas. Synthetic benchmark based on health care example. Measuring throughput in messages per second of different queries. Vary number of schemas.
because we still included the transformation step (which here means copying every element from the incoming message into the new message). This shows the overhead of the transformation step. By omitting the transformation step, the transformation approach would have the same performance as ideal for a single schema.

Complex Query

Figure 6.14(b) shows the message throughput for each of the alternative approaches for a complex continuous query. In this case, Query 5 is used, but the results are similar for queries 6 and 7. Obviously, the throughputs are lower in this experiment than in previous experiment because the query is more complex. This experiment confirms some of the observations made in the previous experiment. In particular, MDQ is again the winner and the performance of query rewrite degrades with the number of schemas. However, there are also important differences: The performance of the query rewrite approach degrades sharply. This observation can be explained with an explosion of the size of the rewritten query if multiple and nested path expressions are rewritten using union expressions. At some point (for about eight schemas), the transformation approach outperforms the query rewrite approach. Comparably, the overhead to transform each incoming message is smaller than the effort to execute the (rewritten) continuous query on the message.

Query Mix

Figure 6.14(c) shows the results of the alternative approaches if all seven continuous queries are applied to each incoming message. Obviously, the message throughput is much smaller in this experiment than in the previous experiments. (Note the different scale of the y-axis.) Nevertheless, the general trends are the same. MDQ is again the (close) overall winner. The performance of query rewrite degrades sharply with the number of schemas because all queries are rewritten using union expressions. One new observation of this experiment is that the transformation approach performs almost as well as MDQ. The explanation is obvious because the transformation approach transforms an incoming message only once and this investment amortizes the more work (i.e., the more continuous queries) is performed on the transformed message. With an increasing number of queries (not shown in Figure 6.14(c)), moving more and more towards a data-warehouse-like workload, the transformation approach would finally outperform MDQ; but this depends on how fast a message can be transformed. As said, the transformation approach is favorable for data warehousing.
6.8.4 XMark Benchmark

The following experiments study the message throughput of the different approaches for the XMark benchmark and a varying number of schemas.

Throughput

Figure 6.15 shows the throughputs for transformation, query rewrite, MDQ, and an ideal world for three sample XMark queries: Query 1 is a simple path expression; Query 4 involves several path expressions with predicates; Query 11 involves heavy computation (joins and sorting), but does not involve sophisticated navigation through the message with path expressions. For all three queries, it can be observed that the throughput of MDQ and transformation is (almost) independent of the number of schemas (i.e., the number of mapping rules), whereas the message throughput of query rewrite drops sharply with an increasing number of schemas. The reason is that the size of the queries grows with the number of schemas and that all (potentially relevant) rules are encoded in every query for every message even though only a small subset of rules are applicable to each particular message.

Depending on the kind of query, query rewrite sometimes performs better than MDQ for data integration scenarios with relatively few schemas (e.g., for Query 1 and Query 11 and less than 100 schemas). As a general trend, the more navigation (with path expressions) is involved in a query the better MDQ performs as compared to query rewrite even if only a few schemas need to be integrated. In fact, Queries 1 and 11 represent two of only four queries of the XMark benchmark for which there is a cross-over point between MDQ and query rewrite; these are the only queries in which MDQ does not outperform query rewrite regardless of the number of schemas (Table 6.4, discussed below).

In this experiment, the transformation approach shows poor performance and is significantly outperformed by the other two approaches. The reason is that in this experiment transformation program is recursively comparing and transforming elements, and this generic form of the transformation program hurts the overall performance of the transformation approach. As said, the schema mixes in this benchmark do not allow an optimization of the transformation approach.

Comparing MDQ to an ideal scenario (with no heterogeneity and mapping rules), it can be seen that MDQ typically comes within 50 percent of the throughput; in many cases, it comes even closer (Table 6.4). These results are encouraging because they show that good system performance can be achieved even on heterogeneous and diverse messages.
Figure 6.15: Scalability with the number of schemas. Measuring throughput in messages per second of different XMark queries. Vary number of schemas.
Table 6.4 shows the results of the complete XMark benchmark, varying the number of schemas from 10 to 1000. Overall, the results of Table 6.4 confirm the observations made for Figures 6.15(a) to 6.15(c). Table 6.4 shows the message throughputs of running every query individually, that is, assuming that it was the only query executing against the stream of messages. Furthermore, Table 6.4 shows the message throughput achieved when executing all twenty XMark queries on every message (denoted as \textit{All} in Table 6.4). Obviously, the message throughput is much smaller if all twenty queries are executed than if only one query is executed. Comparing MDQ with ideal in the \textit{All} case, it can be seen that MDQ is quite close to ideal; its throughput is only about 20 percent below the ideal throughput. One reason is that the effort to annotate the messages according to the mapping rules is amortized across queries, when multiple queries are executed on each message. As a result, MDQ becomes more attractive the more queries are executed on a single message. The transformation approach suffers from the high

<table>
<thead>
<tr>
<th>Query</th>
<th>10 schemas</th>
<th>1000 schemas</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MDQ</td>
<td>QR</td>
<td>TR</td>
</tr>
<tr>
<td>Q1</td>
<td>4014</td>
<td>4688</td>
<td>0.31</td>
</tr>
<tr>
<td>Q2</td>
<td>1699</td>
<td>1700</td>
<td>0.30</td>
</tr>
<tr>
<td>Q3</td>
<td>833</td>
<td>528</td>
<td>0.31</td>
</tr>
<tr>
<td>Q4</td>
<td>910</td>
<td>500</td>
<td>0.31</td>
</tr>
<tr>
<td>Q5</td>
<td>4189</td>
<td>2301</td>
<td>0.30</td>
</tr>
<tr>
<td>Q6</td>
<td>553</td>
<td>137</td>
<td>0.31</td>
</tr>
<tr>
<td>Q7</td>
<td>310</td>
<td>58</td>
<td>0.31</td>
</tr>
<tr>
<td>Q8</td>
<td>29</td>
<td>12</td>
<td>0.30</td>
</tr>
<tr>
<td>Q9</td>
<td>24</td>
<td>10</td>
<td>0.31</td>
</tr>
<tr>
<td>Q10</td>
<td>46</td>
<td>36</td>
<td>0.30</td>
</tr>
<tr>
<td>Q11</td>
<td>25</td>
<td>31</td>
<td>0.31</td>
</tr>
<tr>
<td>Q12</td>
<td>63</td>
<td>82</td>
<td>0.30</td>
</tr>
<tr>
<td>Q13</td>
<td>1093</td>
<td>703</td>
<td>0.30</td>
</tr>
<tr>
<td>Q14</td>
<td>236</td>
<td>111</td>
<td>0.31</td>
</tr>
<tr>
<td>Q15</td>
<td>3158</td>
<td>1627</td>
<td>0.31</td>
</tr>
<tr>
<td>Q16</td>
<td>2021</td>
<td>1413</td>
<td>0.31</td>
</tr>
<tr>
<td>Q17</td>
<td>1032</td>
<td>875</td>
<td>0.31</td>
</tr>
<tr>
<td>Q18</td>
<td>1354</td>
<td>1058</td>
<td>0.27</td>
</tr>
<tr>
<td>Q19</td>
<td>279</td>
<td>144</td>
<td>0.27</td>
</tr>
<tr>
<td>Q20</td>
<td>765</td>
<td>406</td>
<td>0.27</td>
</tr>
</tbody>
</table>

All | 5.6 | 3.5 | 0.32 | 5.6 | 0.04 | 0.003 | 7.7 |
Table 6.5: Throughput [msg/s], XMark Query 1, different MDQ variants.

<table>
<thead>
<tr>
<th>MDQ variant</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager, No mapping rule index</td>
<td>0.77</td>
<td>0.077</td>
<td>0.0079</td>
</tr>
<tr>
<td>Eager, With mapping rule index</td>
<td>571</td>
<td>592</td>
<td>581</td>
</tr>
<tr>
<td>Just-in-time, With mapping rule index</td>
<td>4014</td>
<td>4010</td>
<td>4079</td>
</tr>
</tbody>
</table>

cost of (recursively) transforming the message even when executing all queries on the transformed message. (In Table 6.4, query rewrite is denoted as $QR$, transformation as $TR$. Ideal denotes the perfect world with no heterogeneity.)

**Why does MDQ perform so well?**

We used the XMark benchmark to study why MDQ shows good and constant performance with an increasing numbers of schemas. Table 6.5 shows the throughput of MDQ for XMark Query 1, a varying number of schemas (10, 100, and 1000), and three different MDQ variants. First, MDQ with an eager approach to applying rules and without the YFilter index for the mapping rules shows extremely poor and decreasing performance. Eager means annotating the whole message with new access paths in advance, before query processing begins. Not having an index for mapping rules means searching through all mapping rules for any matching mapping rules while annotating the data. With increasing number of schemas, the number of mapping rules that need to be searched increases, and therefore the throughput decreases. Adding an index for mapping rules removes the burden of searching through mapping rules; thus, the performance of eagerly applying mapping rules remains constant (and much better in total) with increasing numbers of schemas. Finally, applying mapping rules just in time boosts the performance even further. That is, because the initial step of annotating the whole message in advance is removed and only those elements are annotated that are needed to generate the query result. In summary, the good performance of MDQ is because of a nice interplay of applying annotations just-in-time and using an index of mapping rules. (Although not shown in Table 6.5, a just-in-time variant without a mapping rule index would also be possible. In this case, during query processing, all mapping rules need to be searched for any matching mapping rules, which negatively impacts performance.)
Vary Message Size

In this experiment we vary the size of the input message. Figure 6.16 shows the throughputs of all approaches for XMark Query 1, with 100 schemas, and a varying message size. That is, the XMark data generator was configured to produce messages of different size in this experiment. As can be seen, the message size has no impact on the trade-offs between transformation, query rewrite and MDQ. In this particular case, MDQ outperforms query rewrite and transformation independent of the message size. These results were confirmed by experiments made with the other XMark queries and TPoX queries: In general, message size is not a critical parameter to assess the trade-offs of the alternative data integration approaches. (In all our benchmarks, we did not index the data. While indexing may benefit both traditional approaches, it is an interesting avenue for future work to materialize and index annotations of the MDQ approach.)

6.8.5 TPoX Benchmark

The experiments with XMark and the synthetic benchmark studied the performance of the alternative approaches in a data streaming environment. This section gives initial performance results for a more traditional database scenario (without indexes on the data). We use the TPoX benchmark [89], which was designed to evaluate the performance of an XML database. Furthermore, in this experiment we study the effect of caching annotations between query runs.

Table 6.6 shows the response time in milliseconds of running the seven TPoX
Table 6.6: Response time [ms], TPoX benchmark, 100 schemas.

<table>
<thead>
<tr>
<th>Query</th>
<th>Ideal</th>
<th>MDQ (hot)</th>
<th>MDQ (cold)</th>
<th>QR</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPoX Query 1</td>
<td>54</td>
<td>101</td>
<td>148</td>
<td>1261</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 2</td>
<td>39</td>
<td>47</td>
<td>55</td>
<td>510</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 3</td>
<td>14</td>
<td>14</td>
<td>19</td>
<td>45</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 4</td>
<td>89</td>
<td>94</td>
<td>124</td>
<td>869</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 5</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>43</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 6</td>
<td>49</td>
<td>49</td>
<td>66</td>
<td>498</td>
<td>≈ 10min</td>
</tr>
<tr>
<td>TPoX Query 7</td>
<td>78</td>
<td>118</td>
<td>152</td>
<td>1339</td>
<td>≈ 10min</td>
</tr>
</tbody>
</table>

benchmark queries with 100 schemas using the transformation (TR), query rewrite (QR), and MDQ approaches; and in an ideal world (no heterogeneity). For MDQ, two scenarios were studied:

MDQ (cold): The TPoX database is stored in Saxon’s storage manager, but no mapping rules have been applied yet. (That is, no additional edges or nodes have been added to the tree representation of the documents.)

MDQ (hot): The TPoX database is already fully annotated with additional edges and nodes after applying all relevant rules. This was achieved by running the whole benchmark twice and measuring the second run only.

Comparing MDQ to query rewrite, the results of this experiment confirm all the observations of the previous experiments: MDQ clearly outperforms query rewrite, in some cases by as much as an order of magnitude. Furthermore, MDQ is close to an ideal world without any heterogeneity. Comparing MDQ (cold) and MDQ (hot), it can be seen that the overhead of applying the mapping rules and adding edges and nodes to the MDQ data model is roughly 20% in this experiment. That is, caching annotations between query runs brings an additional performance improvement of 20%. (The improvement varies for the individual queries.)

In this setting the transformation approach is impractical because transforming the whole database before processing a query takes about 10 minutes for the 200 MB TPoX database. Table 6.6 lists this approach only for completeness. Once the database has been transformed to single schema, runtimes of subsequent queries would be similar to ideal.

The performance of MDQ (hot) for queries 1 and 7 is much worse than ideal, while for all other queries the performance is almost the same. The reason is that queries 1 and 7 select an “order” element within the TPoX data set. The majority of elements of the TPoX data set are order elements. Therefore, many more elements...
are traversed for queries 1 and 7 than for the other queries. Consequently, many more annotations, which map order elements of different schemas, are present in the annotated data set. MDQ (hot and cold) traverse many more edges than ideal, which leads to the worse performance of MDQ. All other queries do not query order elements. Hence, less annotations are present and the difference in performance between ideal and MDQ (hot) is not big.

6.9 Related Work

Data integration is one of the grand challenges of information technology. Correspondingly, there has been a great deal of previous work on which our work is based. One of the most important research problems addressed in this field is the discovery and design of mapping rules. This research area has matured lately and we build upon the results of that research. Specifically, research projects such as Clio \cite{84, 40} have provided extensive integration frameworks covering both relational and XML data. Products leveraging these technologies include the Clio-based InfoSphere Data Architect \cite{51} and the design editor of BEA’s AquaLogic Data Services Platform \cite{16}. Lately there has even been work on a benchmark for mapping tools \cite{4} from which we adopted basic mapping scenarios discussed in Section 6.3. Again, our work builds on top of the work in this area: Our work assumes that the mapping rules are already known and the goal is to process queries as efficiently as possible in the presence of these mapping rules.

Given a set of mapping rules, there are two ways to process queries: (1) federation with the help of query rewrite and (2) data warehousing based on transformations (i.e., so-called extract-transform-load (ETL) processes) \cite{114}. In the research literature, the federation approach and, in particular, query rewrite has gained a great deal of attention. For declarative LAV (local-as-view) mappings, query rewriting is based on an approach called answering queries using views \cite{54, 76}. Query rewriting for LAV relational schemas has been considered in MiniCon \cite{98} which evaluates the scalability of the rewriting algorithm for queries over a single (global) schema using large numbers of mappings (views). For XML data (and combinations of relational and XML data), there are several proposals for query rewriting \cite{93, 120}. Again the assumption in evaluation is that queries are written against a single (global or target) schema, though the rewriting may use potentially large numbers of mappings or views. These solutions show the scalability of the rewriting process itself (with respect to the number of views), but they do not address the performance of actually executing the rewritten queries. Improving the runtime performance of queries is the primary goal of our work and that is where MDQ, our novel approach, excels. Still, techniques from the work in this
area can be used to optimize MDQ at \textit{compile time} by only creating an index of mapping rules that are needed by the queries (which can be statically determined by query rewrite techniques), and, thus, reducing the size of the NFA index. This optimization is part of future work.

Query rewrite is also the standard technique used for information integration in peer-to-peer systems like Piazza \cite{55} and pay-as-you-go systems like iTrails \cite{112}. In Piazza, queries may be written on any schema and are answered using data in a set of networked data sources using query rewriting and mapping composition. Piazza uses GLAV mappings for which query answering may be undecidable. The main contribution of iTrails is a coloring algorithm in order to deal with transitivity and cyclic mapping rules.

For data warehousing, all major database vendors provide a number of tools to support the ETL process. Example products are Informatica PowerCenter \cite{61} and IBM InfoSphere Information Server \cite{60}. The research community has mostly focussed on the problem of maintaining materialized views \cite{47}. Again, this problem has been studied for both relational data \cite{122} and XML/semi-structured data \cite{3}. Nowadays, there are different approaches aiming at increasing freshness of data in data warehouses. Real-time ETL \cite{21} tries to satisfy freshness requirements without losing the benefits of data warehousing; while On-Demand ETL \cite{108} applies an ETL process only for a subset of the source data, only when a warehouse application asks for it. We believe that MDQ can greatly benefit from the work in this area when materializing and indexing annotations.

The Semantic Web also provides tools for data integration. RDF \cite{102} and OWL \cite{80} are ways to specify mappings between concepts in an open way. In some sense, RDF and OWL are more powerful than the mapping rules traditionally considered for database integration; for instance, OWL supports negation. On the other hand, RDF and OWL are less powerful because they do not allow the specification of complex mapping functions such as the concatenation of \textit{first name} and \textit{last name} or the \textit{convert-date-from-zh} function from our motivating example (Section 6.2). It is an important avenue for future work to study how the query processing techniques for data integration studied in this work can be extended to support OWL features such as negation.

This work studied the performance of the three alternative approaches to data integration in the context of continuous processing of input messages; thus, essentially evaluating continuous queries on data streams. There has been a great deal of work in the area of data stream processing; for example, \cite{107, 28, 26, 86, 1}. This work could be leveraged to implement a \textit{heterogeneous} data stream management system with just-in-time data integration. To the best of our knowledge, there has not been any prior work that specifically studies the processing of continuous
queries on heterogeneous data streams.

The term just-in-time data integration is adopted from the programming language community. In the programming language community, the term just-in-time (or JIT) refers to techniques that optimize programs at the latest possible moment at runtime, leveraging as much information as possible. In the database community similar ideas have been pursued for data cleansing [101] and query optimization with Eddies [7].

6.10 Conclusion

This chapter presented just-in-time data integration and Mapping Data to Queries. Just-in-time data integration is the principle to integrate data at the latest possible point in time—at runtime of the query. Mapping Data to Queries is an implementation of just-in-time data integration. The idea of MDQ is to interweave query processing and data integration: during query processing mapping rules are applied by annotating the data with additional access paths. The advantage of MDQ over traditional approaches (the transformation approach and the query rewrite approach) is that it can optimize the integration based on both, the query and the data. Specifically, MDQ does not invest in integrating data that is not relevant for any query. In this way, MDQ potentially outperforms the transformation approach. Furthermore, MDQ only applies mapping rules that are relevant to the data, rather than considering the union of all mapping rules. In this way, MDQ potentially outperforms the query rewrite approach.

With the help of performance experiments, we showed that MDQ indeed outperforms both traditional approaches to data integration. Our experiments were based different benchmarks including a synthetic benchmark, XMark, and TPoX; and covered a range of workloads involving documents with mixes of schemas, queries with varying complexity, and varying numbers of schemas. In all settings, we showed that MDQ scales well with the number of schemas.

Although MDQ scales with number of schemas, it is important to keep in mind that MDQ is not a one-size-fits-all solution. The transformation approach is still the right solution for classic data warehouse workloads where there are many bulk queries and low data freshness requirements. Query rewrite is a crucial optimization technique that can be used for the optimization of MDQ and any data integration approach. Furthermore, query rewrite outperforms MDQ for data integration scenarios with few schemas and mapping rules. As a consequence, our next step is to explore a data integration system that leverages all three techniques and decides which technique to use based on workload characteristics, bridging the gap between federation-based approaches and data warehousing.
Table 6.7: Expressing the remaining basic mapping scenarios of the STBenchmark [4] using mapping rules. (This table completes Table 6.1 on page 102.)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Source and target schema</th>
<th>Mapping rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self joins</td>
<td></td>
<td><code>s:gene* → t:gene*</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:name → t:name</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:protein → t:protein</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:gene[s:type='primary'] → t:gene</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:name → t:name</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:protein → t:protein</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:gene[s:type!='primary'] as $g → t:synonym</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(t:gid){for $t in /t:gene where $t/t:protein = $g/s:protein return $t/t:name/text()}&lt;t:gid&gt;</code></td>
</tr>
<tr>
<td>Denormalization</td>
<td></td>
<td><code>s:emp as $e → t:emp</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(t:dept){for $d in /s:department where $d/s:id = $e/s:deptid return $d/s:name/text()}&lt;t:dept&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:name → t:name</code></td>
</tr>
<tr>
<td>Object fusion</td>
<td></td>
<td><code>s:experiment as $e → t:experiment</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{for $s in /s:sample where $s/s:key = $e/s:key return $s/s:data}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:key → t:key</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:data → t:data</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s:sample[s:key=/s:experiment/s:key] → t:experiment</code></td>
</tr>
</tbody>
</table>

\(^a\)Records in the target are associated using foreign-key relationships.

\(^b\)Records of a source are merged, values are stored redundantly.

\(^c\)Records with same key are merged.
Chapter 7

Holistic Data Integration & Integration Independence

In this chapter, we present two concepts in the field of data integration: *holistic data integration* and *integration independence*. Holistic data integration allows to perform both schema-level and data-level integration within a single system. Integration independence allows to move transparently between a materialized view and a virtual view of the integrated data. We will show how Mapping Data to Queries supports these two concepts.

Parts of the contents of this chapter have been published at the International Conference on Conceptual Modeling 2009 [50] and the International Workshop on New Trends in Information Integration 2010 [53].

7.1 Introduction

Data integration is best thought of not as a single act, but as an iterative process [49]. Typically, the individuals doing the integration are not experts in all of the data; but must first understand what data is available, how good it is, and how to represent the data in the application. A plan for integrating the data must be prepared, and only then can they move from design to execution, and actually integrate the data. Once the integration takes place, users often discover problems or the needs may change, and they have to go through the whole process again to iteratively refine it. There are different tools for different (overlapping) parts of this process, as well as for different needs. The concepts of this chapter try to simplify this process by bridging the gaps between today’s fragmented tool space.
Figure 7.1 outlines today’s tool space. There are two dimensions with different (overlapping) tools. First, there are different tools for different stages of the integration process. There are tools for schema-level integration (e.g., Clio [40] and MDQ) and data-level integration (e.g., specifying a same-as relationship in RDF [102]). Second, there are different tools that provide either a virtual view of the integrated data (e.g., Minicon [98] and answering queries using views [54, 76]) or a materialized view of the integrated data (e.g., products supporting ETL such as Informatica PowerCenter [61] and IBM InfoSphere Information Server [60]). Figure 7.1(a) shows that today’s tools for data integration are too fragmented resulting in an integration process that is too time-consuming and too complicated.

We need to go beyond the status quo, towards a simplified process for information integration. We need to extend today’s tools to cover both data-level and schema-level integration tasks, and we need to make the design process (and the applications) independent of the actual integration method. We call the first of these holistic data integration, and the second integration independence.

**Holistic Data Integration.** Mapping Data to Queries, which was presented in the previous chapter, only deals with schema-level relationships between a data source and a target. Today, other tools are needed to handle data-level integration
tasks. Such tasks include entity resolution, which identifies entities in a data source that may represent the same real-world object, and data fusion, which creates a consistent, cleansed view of data from potentially multiple conflicting representations. There is little support for iteration between schema-level and data-level tasks in the integration process. This is unfortunate, because there is no perfect ordering of the tasks. Sometimes, mapping can help with understanding the data and hence with entity resolution and data fusion. But those tasks can also provide valuable information to a mapping process. By handling both schema and data-level tasks in a common framework, holistically, we enable easier iteration among these phases, and hence, a smoother integration process.

**Integration Independence.** There are two radically different integration methods: virtualization and materialization [114]. Virtualization (aka, query rewrite) leaves the data where it is, as it is, and dynamically integrates it on request. Materialization (aka, transformation) integrates the data up front, creating a new data set for requests to run against. Each has its strengths. Virtualization always gets the freshest data, and does no unnecessary work, since the data is integrated only if needed (a lazy form of integration). Materialization often provides better performance, but may integrate data that will never be requested (an eager approach). Often, the best solution will require a combination of these two approaches. In fact, virtualization cannot solve the whole integration problem today, as we simply do not understand how to do much of integration, including data fusion and entity resolution, virtually. The materialization process handles these data-specific tasks, but it is too heavy duty for some use cases, and a materialization often takes too long to design and build. The decision of which approach to use, and when, must be made early in the integration design process, and, as different integration tools must then be used for the different pieces, is difficult to change. Ideally, applications should be independent of how, when, and where information integration takes place.

While progress may be made on holistic data integration and integration independence separately, together they hold the potential for truly radical simplification. It would clearly be a leap forward to have a single engine that could move seamlessly between virtualization and materialization, with no changes to the application program, and at the same time can tackle both schema and data-related integration issues (Figure 7.1(b)).

In the following, we will show how Mapping Data to Queries can be extended to support holistic data integration (Section 7.2) and integration independence (Section 7.3).
7.2 Holistic Data Integration with MDQ

Holistic data integration is the concept of performing schema-level and data-level integration within a single framework. In the previous chapter, we introduced Mapping Data to Queries (MDQ), which allows to integrate data at the schema level. MDQ supports mapping rules that define relationships between a source and a target schema. In the previous chapter, mapping rules were of the general form $source-expr \rightarrow target-expr$, which map entities of the source schema to entities of the target schema. We showed that MDQ applies these mapping rules by annotating the data with additional access paths at the runtime of a query. To perform data integration at the data level, we first discuss a mapping rule that merges data instances from different documents and then we show how the idea of annotations can be extended to support such mapping rules.

To perform data integration at the data level, we add a new type of mapping rule that merges instances of data. Mapping rules that merge two instances of data are of the form $instanceId_1 \leftarrow instanceId_2$. This specifies that the data instance with identifier $instanceId_2$ should be merged into (be seen as part of) the instance identified by $instanceId_1$. This creates an asymmetric merge into semantics. Other semantics are definitely possible. For example, the instances could be merged into a new instance (symmetric merge semantics), or not merged at all, but treated as one during query processing (equivalence semantics) so that only one is returned.

From an implementation perspective, merge into is simpler to model, and seems to offer sufficient power for the scenarios we have worked with so far, but more investigation is clearly needed.

The key idea of Mapping Data to Queries—annotating data with additional access paths—can be adopted to data-level integration as well. In particular, annotations can be used to apply data-level mapping rules of the form $instanceId_1 \leftarrow instanceId_2$. The idea is to annotate the internal tree representation of the data with a composite node that contains the original nodes identified by $instanceId_1$ and $instanceId_2$. The composite node has exactly one identifier, that of the left node of the mapping rule (i.e., $instanceId_1$). Figure 7.2 illustrates an annotation of a composite node. In Figure 7.2, the composite node is represented by the bold dashed line, its identifier is #1. The new composite node keeps all of the inbound and outbound edges of the original nodes.

The composite node will be handled by the query processor in the same way as any original node. Because it is possible to reach the composite node twice during query processing (over different inbound edges with the same name), duplicate elimination is always needed. All queries require an additional duplicate elimination step. Duplicate elimination is based on the composite node’s identifier, and
we can reuse the duplicate elimination algorithm of the query processor.

The disadvantage of this approach is that a merged instance of an element may violate the element’s original schema. For example, if two person elements are merged that each have one child element name, the merged element now has two children name. As a solution, one could specify another data-level mapping rule that merges these name children into one child, or queries that process the data have to be aware of such situations (e.g., by always requesting only the first name child of a person).

MDQ supports both schema and data-level integration at the same time—thus, implementing holistic data integration. Both types of annotations, additional access paths and composite nodes, work well together. Figure 7.3 illustrates both types of annotations used at the same time. (The documents and schemas used in Figure 7.3 are explained in detail in the next chapter.) In Figure 7.3 there are two schema-level mapping rules (\texttt{w:table} $\rightarrow$ \texttt{p:person} and \texttt{d:author} $\rightarrow$ \texttt{p:person}) and one data-level mapping rule (#2 $\leftarrow$ #8) which are being applied. Thus, there are two additional access paths (represented by the bold arrows in Figure 7.3) and one composite node (represented by the bold dashed line). A query \texttt{//p:person}, which returns all \texttt{p:person} elements, would now return exactly one \texttt{p:person} element (by following both new access paths and using duplicate elimination to return the composite node only once).

In summary, we extended the annotations of Mapping Data to Queries to perform data-level integration. At the data level, we use composite nodes to merge instances of data in different documents. The combined used of additional access paths and composite nodes enables data integration at the schema and data level holistically.
7.3 Integration Independence with MDQ

Integration independence is the concept of moving transparently between a virtually integrated view and a materialized integrated view of the data. As presented in Chapter 6, Mapping Data to Queries adds annotations virtually, at the runtime of a query. Annotations are only added if needed. However, in Section 6.8.5 we already showed the positive effect of caching annotations. That is, annotations were not thrown away after query processing, but cached and reused when processing the next query. This, already, is a first step towards moving from a virtual view to a materialized view of the integrated data. Caching annotations is completely transparent to the application (i.e., the query). The query does not know whether annotations are being added at runtime or already existed before the start of query processing. The idea of integration independence is to move to a completely materialized view of the integrated data; that is, a transformed document in the target schema.

Transforming an annotated document in MDQ to a document in the target schema is quite simple. The idea is to validate the document against the target schema. Validation of an XML document typically traverses the whole document to check if this document is compliant with the target schema, but it also adds types to the nodes of the document (e.g., an untyped atomic element is being typed as a string). Similarly, validating an annotated document in MDQ traverses the document and, in addition to typing, removes any unnecessary access paths that are not needed in the target schema. The remaining access paths are then materialized (e.g., written to disk). Removing unnecessary access paths transforms the original document in the source schema to a document in the target schema.

Figure 7.3: Holistic data integration with MDQ. Both types of annotations (additional access paths and composite nodes) work well together.
Figure 7.4 illustrates how MDQ supports integration independence. The document in Figure 7.4(a) is present in the source schema only; additional access paths are added virtually if needed. Figure 7.4(b) caches annotations that were previously introduced by mapping rules. Caching annotations is a first step towards a materialized view. The document in Figure 7.4(c) is present in the target schema. Via schema validation, unnecessary access paths (w:table, w:tr, w:th, and w:td) have been removed. Going back from a materialized view to the virtual view is possible by simply throwing the materialized view away, and applying mapping rules virtually to the original document again. The application on top of MDQ is at all times unaware of the form in which the data is present.

It is also possible to materialize a (virtual) composite node, which was introduced by a data-level mapping rule described above. The composite node is removed and replaced by a standard node, and only one parent is kept. Here, we make use of the asymmetric merge into semantics. The node on the left-hand side of a mapping rule $\text{instanceId1} \leftarrow \text{instanceId2}$ is given preference to keep its parent node. Figure 7.5 illustrates the materialization of the composite node of Figure 7.2. The composite node is replaced by a normal node and the left parent edge “a” is kept. All other incoming edges are removed. The materialization of both types of annotations, additional access paths and composite nodes, shows that holistic data integrating and integration independence work well together in MDQ.
7.4 Conclusion

In this chapter we presented two new concepts in the field of data integration, holistic data integration and integration independence. These concepts bridge the gaps between different tools for schema-level and data-level integration as well as virtualization and materialization. Today’s fragmented tool space can be radically simplified when implementing both concepts in a single framework (Figure 7.1(b)).

We implemented both concepts in Mapping Data to Queries. To support holistic data integration, we implemented composite nodes for data-level integration. Together with additional access paths for schema-level integration, they enable a holistic data integration in MDQ. The implementation of composite nodes has been demonstrated in the People People People application (see next chapter). To support integration independence, we implemented caching of annotations and removing of unnecessary edges from an annotated tree structure. As studied in Section 6.8.5, caching annotations results in a 20% performance improvement. Executing queries on a materialized view without unnecessary edges yields the same runtime performance as having only one schema in the first place; and, thus, the performance is close to an ideal world without heterogeneity (as shown in Section 6.8.3, figures 6.14(a) to 6.14(c)).
Chapter 8

The People People People People Application

In this chapter, we present People People People, an application on top of Mapping Data to Queries (MDQ). The application integrates person-related data from DBLP [77] and Wikipedia, showing how MDQ can be employed in practice. Furthermore, it demonstrates a use case of the concept of holistic data integration presented in the previous chapter.

This application was part of our demonstration of MDQ at the 2010 VLDB conference [58].

8.1 Introduction

Setting up a data integration system typically involves an iterative process of exploration and integration. In the beginning, the user explores the data to get familiar with its content and structure. This enables him to integrate a first chunk of data. He then proceeds to further explore the same or additional data sources, eventually expanding and refining the integration. Data integration systems should be flexible enough to support such an iterative process. Furthermore, today’s businesses deal with hundreds of schemas containing similar data, as for example in Health Level 7 [56]. Thus, data integration systems should also be scalable in the number of schemas.

Our novel data integration system, Mapping Data to Queries, offers great flexibility and scalability. Our approach is flexible in that it executes schema-level and data-level integration tasks in a single framework. As we will demonstrate, this allows
to integrate data holistically in an incremental and iterative fashion. Furthermore, MDQ scales well with the numbers of schemas (see Chapter 6).

In order to demonstrate Mapping Data to Queries, we have created an application that highlights the features of our data integration system. We call the application People People People. It lets users integrate person-related data as they explore. More precisely, it allows users to search for people, display information about people, and browse through a network of related people, while at the same time refining the integration incrementally, both on the schema level and data level.

People People People is inspired by the community portal DBLife [37]. DBLife is an application on top of Cimple [36]. Cimple focuses on automatically generating mapping rules and employs an ETL approach for data integration. Our application is meant to demonstrate the functionality of MDQ. We do not automatically generate mapping rules.

8.2 Data and Schemas Used

The People People People application integrates data about people (in particular computer science researchers) from DBLP [77] and Wikipedia. In both data source, data is available as XML. Figure 8.1 shows sample XML snippets from Wikipedia and DBLP. The schemas of these documents are illustrated in Figure 8.2. The third schema in Figure 8.2, People, is the schema used in our application. There are schema-level mapping rules that map the DBLP and Wikipedia schemas to the People schema. Mapping rules are created before the start of the demonstration. A small sample of mapping rules is shown in Figure 8.3. A (semi-)automatic schema mapping tool could be used to create this initial set of mapping rules, though ours are hand-written. On the data level, there are scientists that are present in

Figure 8.1: Data snippets from Wikipedia and DBLP.
both data sources, in DBLP as well as in Wikipedia. For most scientists there are multiple data entries in DBLP. These data entries are merged using data-level mapping rules. Since DBLP uniquely identifies researchers [77], we were able to create an initial set of data-level mapping rules. More mapping rules are generated whenever the user of our application merges results (see demonstration scenarios below).

As discussed in chapters 6 and 7, Mapping Data to Queries annotates data with additional access paths and composite nodes at the runtime of a query. Figure 8.4 shows the fully annotated trees of the documents of Figure 8.1 after all mapping rules have been applied. Mapping rule R1 adds a p:person annotation to the labeled tree representation of the left document of Figure 8.1 (when the w:table edge is traversed for the first time). The annotation is represented by the bold edge with the label p:person in Figure 8.4. Mapping rule R2 specifies that the nested structure w:tr/w:th should be unnested (flattened) into p:name. In Figure 8.4, this rule introduces an annotation (labeled p:name) that points from w:table directly to w:th. Mapping rule R3 constructs a p:name child containing the text value of the original author (through the variable binding $a$). Mapping rule R4 in Figure 8.3 merges the node with id #8 into node #2. Whenever either of these nodes is accessed for the first time, the data is annotated with a composite node that contains the original nodes.

Given the annotated trees in Figure 8.4, it is possible to answer a query written in the People schema. For example, a query //p:person (which uses the People schema) would return the p:person element of Hector-Garcia Molina once.15

---

15As discussed in Chapter 6 the result of query //p:person is serialized as w:table and not as w:tr/w:th → p:name

Figure 8.2: Source and target schemas.

R1: w:table → p:person
R2: w:tr/w:th → p:name
R3: d:author as $a$ → p:person (<p:name>{$a/text()}</p:name>)
R4: #2 ← #8

Figure 8.3: Mapping rules.
8.3 Application

People People People is a web application. The web application consists of a search bar, an information pane with a merge button, and two additional tabs Rules and Query. The search bar that allows users to search for people by name. (Each search translates into an XQuery query under the covers.) Search results are displayed on the information pane as little icons attached with an identifier and name, which users can click on (Figure 8.5(a)). Clicking on an icon displays information about the person as a list (Figure 8.5(b)). Double-clicking on an icon of a person results in an overview of related people and what that relationship is (Figure 8.5(c)). The merge button merges multiple selected persons. (The click of the merge button translates into adding a new data-level mapping rule under the covers.) To select multiple persons, the user holds down the shift button while clicking on the persons’ icons. The Rules tab lets users view and edit all mapping rules stored in the system. This allows the user to refine current mapping rules, to delete mapping rules in case of mistakes (e.g., false merges), or to incorporate additional data sources. The Query tab lets users post individual queries as well as modify existing queries used by the web application. In summary, by interacting with the web application, users can search and browse through a network of people, p:person, and as discussed in Chapter 7 the composite node now has two p:name children. Our application is agnostic to both issues by always creating new result elements and querying the first child only, e.g., for $n in //p:person/p:name[1] return <name>$n/text()</name>, which returns a list of names of all (merged) persons.
while at the same time refining the integration incrementally, both on the schema level and data level.

Our demonstration system consists of a client and a server. The client runs People People People as a web application in a standard browser. The application connects to the server through remote procedure calls. It sends XQuery queries to the server and the server answers with XML. The application presents the results as figures and text. Clicking on a figure or on the text generates a new query, which is in turn sent to the server and answered quickly to provide an interactive user interface.

The server runs MDQ. The MDQ engine is loaded with the pre-defined mapping rules. Two types of data access are supported, standard file input/output to local storage, and connections to web servers on the internet over standard HTTP. Local storage consists of a directory structure containing data files, indexes, and meta data, and is loaded with parts of the DBLP data set, available in XML. The DBLP data set is stored on the server without any modifications to the data. In particular, it has not been transformed to any other schema. In the demo, the engine will retrieve Wikipedia web pages in an on-demand fashion. The web pages are returned by Wikipedia in the XHTML format—a variant of XML. Therefore it is possible to process these web pages using standard XQuery processing techniques.

8.4 Demonstration Scenarios

The scenarios below deal with three methods of interaction with the system, ranging from simple to complex. They demonstrate finding information about people and displaying their relationships, manually merging duplicate data records, and integrating a third new data source by adding more mapping rules.

**Scenario 1: Searching and Browsing Information.** A fresh PhD student wants to know more about famous scientists in his area of research. He logs on to the People People People web application and searches for a researcher he has heard of: Hector Garcia-Molina. The system returns three results, displayed as small icons tagged with name and id (Figure 8.5(a)). The student clicks on the last result. The system provides the student with additional, structured information about this person (Figure 8.5(b)). The student further wants to know what people Hector Garcia-Molina is related to. He double-clicks on the last search result. The system displays the data record of Hector and four persons Hector is related to. The related persons are again shown as small figures tagged with name and id. Furthermore, there are labels in between Hector and each of the related persons.
stating the type of relationship (Figure 8.5(c)). For example, Gio Wiederhold was Hector’s doctoral advisor. Clicking on any of the related persons allows the student to display information about them. Double-clicking on one of the persons will result in displaying their relationships. In this way, the student is able to browse through a network of related scientists.

**Scenario 2: Merging Duplicate Records.** Judging from the additional information in two of the data records found for Hector Garcia-Molina, the PhD student decides that these two data records refer to the same real world person with name Hector Garcia-Molina. The student wants to add this knowledge to the system. He therefore selects both data records and clicks the merge button. Immediately, the system updates the view to display only one data record instead of the previous two records. Internally, a new data-level mapping rule has been added to the system. The student may continue to use the merged data record as described in Scenario 1.

Note that figures 8.5(b) and 8.5(c) already display information from such a merged data record. In Figure 8.5(b) the merged data record’s identifier shows the identifiers of the individual records #2.105 and #0.362. The data items: fields, in-
stitutions, and residence stem from Wikipedia while the publication data stems from the DBLP data set. Similarly, in Figure 8.5(c), the relationships doctoral advisor and doctoral student are retrieved from Wikipedia while the co-author relationships stem from DBLP.

**Scenario 3: Adding a New Data Source.** After a while the PhD student thinks that there is important information missing. He wants to incorporate an additional data source. He therefore opens the Query tab to add a look up of Amazon webpages to the existing search query, so that data from Amazon will be included in the results. The student then queries for Hector again. No additional data appears in the result view. The student recognizes that he must map the Amazon schema to the People People schema. Therefore he opens the Rules tab and adds the schema level rule $a:h1 \rightarrow p:person$ (let $a$: be the namespace of Amazon). This rule views the webpage (the headline of the webpage) as a person. As the student queries for Hector again, he will notice an additional data record. Yet, this additional record is empty. The student goes back to add another mapping rule to include all books of people as additional information. Revisiting the last search result of Hector, the student finds that all of Hector’s books show up in the information box. Now, he merges this record with the existing records for Hector. The aggregated data record now contains information gathered from DBLP, Wikipedia, and Amazon.

In summary, the PhD student was able to include a new data source incrementally. Iteratively, he mapped information from that source to the People People People web application schema. Finally, he merged data from the new source with data that already existed in the database.
Chapter 9

Conclusion

This thesis studied two current trends in the world of information: (1) more and more data is shared in real time, and (2) data is becoming increasingly diverse. Both trends are still ongoing and require scalable systems that can keep up with future workloads.

The first trend is that more and more data is shared in real time. This trend is accelerated by social networks such as Twitter and Facebook, whose success is mainly based on the distribution (and analytics) of information in real time. Consequently, the first part of this thesis studied systems that efficiently distribute information in social networks, and systems that analyze data in real time with strong consistency guarantees.

The second trend is that data is becoming increasingly diverse. The reason is that an increasing number of businesses interchange data, yet each business has its own data formats (i.e., schemas) for important data. They will never agree on a single (global) schema. However, there are standards such as Health Level 7 or XBRL that define conventions for interoperability across schemas. This gives businesses the flexibility to define their own schemas while still being able to interoperate. Therefore, the second part of this thesis studied systems that efficiently process interchanged data in many (i.e., thousands) different schemas.

9.1 Summary of this Thesis

In summary, this thesis made the following contributions:

We studied in detail how to build propagational data flows; which support, for example, the distribution of information in social networks in real time. Thereby we evaluated three approaches: (1) a naive approach, which executes all steps
of the data flow synchronously; (2) the state-of-the-art approach, which uses an external queue to execute the data flow asynchronously; and (3) our new integrated approach, which integrates the (asynchronous) execution of data flows into the key-value store. We showed that the naive approach cannot guarantee low response times and cannot handle bursts of inserts. The state-of-the-art approach overcomes these drawbacks but does not utilize resources efficiently. By combining data storage and data flow execution, the integrated approach utilizes resources more efficiently, utilizing 35% less network resources and 15% fewer machines than the state-of-the-art approach. Thereby, our approach performs and scales equally well as the state-of-the-art approach. We conclude that tightly integrating data storage and data processing is beneficial when building systems for propagational data flows.

Based on these findings, we studied how to build a system for real-time analytics. In such a system, updating data incrementally is crucial to achieving good data freshness. Furthermore, a system for real-time analytics should: have a simple, yet expressive programming model; scale with number of nodes of the system (to process more input messages); be tolerant to node failures; and provide exactly-once semantics. Implementing these requirements, we studied important design space options: (1) replicating input data and result data is crucial to providing data availability and non-blocking execution in case of node failures; (2) replicating intermediate data should be avoided as it significantly impacts performance; (3) combining computation and storage is beneficial as discussed above; and (4) distributed transactions should be fine-grained to achieve best performance. These findings led to the creation of Limmat. Limmat is a system that analyzes data in real time and implements all of the requirements mentioned. In particular, it scales well with the number of nodes. In comparison to related systems, Limmat extends Google Percolator by providing exactly-once semantics of task execution, and it extends Twitter Storm by providing strong consistency in case of node failures.

With the help of an application that optimizes online newspapers based on social media, we showed how Limmat can be employed in practice.

To efficiently process interchanged data in many different schemas, we presented Mapping Data to Queries (MDQ). The key idea of MDQ is to integrate data at the latest possible point in time, at the runtime of a query. We call this principle just-in-time data integration. MDQ implements this principle by interweaving query processing and data integration: mapping rules are matched at query runtime and data is annotated with additional access paths. The advantage of MDQ over traditional approaches (the transformation approach and the query rewrite approach) is that it can optimize the integration based on both, the query and the data. In particular, MDQ only integrates data that is accessed by a query, rather than
integrating all data. In this way, MDQ potentially outperforms the transformation approach.\footnote{In addition, the transformation approach employs long-running transformation scripts, which impact performance as well.} Furthermore, MDQ only applies mapping rules that are relevant to the data, rather than always considering a union of all mapping rules. In this way, MDQ potentially outperforms the query rewrite approach. MDQ is particularly effective in unpredictable environments where the schemas of data are not known in advance, and documents consist of mixes of schemas (i.e., some parts of a document correspond to one schema while other parts correspond to other schemas). We showed that MDQ scales well with the number of schemas, and ultimately outperforms both traditional approaches by orders of magnitude in extreme cases.

Furthermore, we introduced two new concepts in the field of data integration: holistic data integration and integration independence. The concept of holistic data integration aims at performing both, schema and data-level integration in a single framework, bridging the gap between different tools for schema-level integration and data-level integration. We showed how MDQ can be extended with composite node annotations that merge data items at query runtime. Together with access path annotations, this enables MDQ to support holistic data integration. The concept of integration independence aims at transparently moving between a virtually integrated view and a materialized integrated view of the data, bridging the gap between federation-based tools (i.e., virtualization) and transformation-based tools (i.e., materialization). We showed that annotations can be cached and reused, and that annotated data can be transformed and materialized. This enables MDQ to move between a virtually integrated view and a materialized integrated view. Thus, MDQ supports integration independence. The interplay between both concepts, holistic data integration and integration independence, could significantly simplify the current landscape of tools for data integration.

Finally, we presented People, an application which runs on top of Mapping Data to Queries. People integrates person-related data from online and offline data sources such as Wikipedia and DBLP. This demonstrates how MDQ can be used in practice.

\section*{9.2 Future Work}

There are several interesting avenues for future work: combining Limmat and MDQ, supporting hotspots in Limmat, and installing MDQ in an industrial setting. First of all, an interesting avenue is to combine both systems, Limmat and Mapping Data to Queries, into a single system. The benefit of combining both systems is
Combining both systems is straightforward. The combined system consists of Limmat as such, plus an installation of MDQ at each node. Figure 9.1 illustrates the combined system. The input to the system is a stream of key-value pairs where each value is formatted as XML; values can come in many schemas and mixes of schemas. Key-value pairs are processed by map and reduce functions, which are implemented not in Java (such as in our experiments in Chapter 4), but in XQuery. The system either outputs results actively, or results can be queried. A central mapping rule repository maps different schemas to the schema expected by the functions. The working principle of this system is then straightforward: Limmat is used to distribute the computation inside the system using transactions and synchronization to execute tasks as usual. The execution of these tasks (in XQuery) is handled by MDQ, which uses the centralized mapping rule repository to integrate data during the execution of tasks. Thus, the combined system scales in two dimensions: with the number of input messages and with the number of schemas of input messages. Implementing this combined system and evaluating its performance is a great avenue for future work.

Concerning the Limmat system on its own, further future work includes the support of hotspots. As discussed, a hotspot (or bottleneck) is a single point of the computation that is overloaded and cannot be scaled trivially. In Limmat, a
hotspot is a task that performs a time consuming operation on the same key too often. The node executing the task is overloaded and cannot execute the task fast enough; the queue which buffers scheduled tasks will at some point exceed its capacity. Supporting hotspots is not trivial because the computation on a single key cannot be broken up into smaller parts. Thus, additional mechanisms to support hotspots are needed. In Chapter 4, we already discussed ideas with respect to how to solve this problem. In particular, we discussed an additional merge function (similar to [119]), with which it is possible to (1) break up the computation into multiple smaller parts, which can be executed on different nodes, and (2) merge the individual results at the end. An experimental study of different ideas that support hotspots is therefore an interesting part of future work.

Concerning Mapping Data to Queries, future work includes installing MDQ in an industrial setting. An interesting opportunity to do so has been suggested by a Swiss-based, globally operating pharmaceutical company. The company has departments in many countries. Each department has its own database management system for managing employee data and customer data. The database schemas of the departments are (naturally) different from each other. This is partly due to legacy reasons and partly due to individual laws in the different countries. Furthermore, data in the databases of different countries may be inconsistent. For example, an employee which has switched teams is present in the database of his/her new team, but is still present in the database of the old team as well. The company is interested in cleaning up the data on the schema and data level through the participation of their employees in all countries. Therefore, they think of a social-network-like web application, which gives employees the power to individually fix inconsistencies in the databases and clean up the data. Their changes will be visible immediately using virtual data integration techniques (such as annotations of MDQ), but the changes will later be propagated to the databases and materialized. Installing MDQ in this setting requires research on how to specify mapping rules easily, given only a web browser; how to extend MDQ to relational databases; and how to “push back” the results of the integration to the database. This is a fascinating avenue for future work, which would also permit the commercialization of the ideas of Mapping Data to Queries.


166


170
Curriculum Vitae
Martin Hentschel

Education

PhD Studies in Computer Science
ETH Zurich, Switzerland
November 2007–March 2012

Master of Science ETH in Computer Science
with distinction
ETH Zurich, Switzerland
March 2006–October 2007

Bachelor of Science ETH in Computer Science
ETH Zurich, Switzerland
October 2004–April 2007

Vordiplom in Informationssystemtechnik
TU Dresden, Germany
October 2002–October 2004

Abitur
Rainer-Fetscher-Gymnasium Pirna, Germany

High School Exchange Year
North High School, Akron, OH, USA
August 1998–June 1999
Work Experience

**Software Development Engineer in Test**  
Microsoft, Redmond, WA, USA  
Internship  
November 2010–January 2011

**Software Engineer**  
ECOFIN Research & Consulting, Zurich, Switzerland  
Internship  
August 2005–October 2005

**Civilian Service**  
Diakonisches Altenzentrum Graupa, Pirna, Germany  
August 2001–December 2001  
AWO Jugendwerkstatt, Heidenau, Germany  
January 2002–May 2002

Academic Publications


