Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Efficient Management of RFID Traceability Data

Diplom-Informatikerin Univ.
Veneta M. Dobreva

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Univ.-Prof. Dr. Torsten Grust
   Eberhard Karls Universität Tübingen

Die Dissertation wurde am 26.03.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03.08.2013 angenommen.

Abstract

Several application fields such as automotive industry, pharma industry, and logistics are increasingly employing Radio Frequency Identification (RFID) technologies to track their goods in business processes. The frequently produced large amounts of events constitute new challenges to modern databases. An efficient data staging process as well as efficient query mechanisms for both – processing of the latest information (OLTP) and processing of analytical queries (OLAP) are required.

In this thesis, different mechanisms for the efficient management of traceability data using the example of RFID are presented. First, we summarize the challenges that RFID data poses to a storage system and examine whether existing relational DBMS approaches fulfill these requirements. The approaches are compared using a mixed benchmark, consisting of concurrent inserts and queries. Further, a novel approach, where the OLTP and the OLAP components reside in the same relational database, is introduced and compared to the existing solutions.

Second, inspired by prior work on RDF triple stores, we present a scalable dedicated system for efficient storage and fast querying of RFID data: the RFID Triple Store. The challenges posed by RFID data are addressed as follows: (1) elaborate indexing techniques leveraging the specifics of the data are incorporated, in order to enable efficient data staging; (2) the query engine takes advantage of the characteristics of the data to speed up query processing. Our experimental studies show that the RFID Triple Store can achieve both a significantly higher insert throughput and a better query performance compared to the state-of-the-art of RFID data management.

Finally, mechanisms for distributed RFID data management are explored. We apply the MapReduce paradigm to conduct distributed query processing and analyze how suitable MapReduce is in an RFID context. Further, a distributed solution using our RFID Triple Store is introduced. We compare both approaches and determine that the distributed Triple Store significantly outperforms the MapReduce implementation.

Contents

List of Figures

List of Tables

1 Introduction

Automatic identification (Auto ID) is the process of automatically tracking physical or logical objects – e. g., people, animals, goods, or invoices. The collected information serves for analysis in different industries – e. g., the behavior of animals or the pedigree of pharmaceutical drugs. Figure 1.1 shows an overview of the most common Auto ID methods [25]. Barcode systems are extremely cheap in production, but have only low storage capacity and cannot be re-programmed. To overcome these disadvantages, a more flexible solution was invented in the 1980's: Radio-frequency identification (RFID) [25].

RFID data is stored in a silicon chip and is transferred without line of sight between the object carrying the data and its reader. Tags (RFID tags are applied to or incorporated into the scanned objects) consist of an integrated circuit for storing information and for modulating and demodulating a radio-frequency signal, together with an antenna for receiving and transmitting the signal. This way, a reader (or sensor) is able to interrogate the stored tag information from a distance of several meters away and beyond a line of sight. Stable international standards as well as the steadily maturing reliability and decreasing costs of equipment cause a significant increase in the adoption of RFID technology. More and more different application fields such as postal package service, aviation industry, health care, and baggage tracing in airports [14, 15, 24, 28, 50] deploy RFID.

Infrastructures of readers enable to accurately track and trace moving objects throughout their life-cycles. For instance, governments apply RFID technology in traffic management and public transit for season parking tickets, for e-tolling on motorways and bridges, for payment in bus, rail and subway, and even in passports. Not least, retailers benefit from RFID in asset and inventory tracking as well as in item-level sales. Touch-free payment systems have been developed, which work with embedded tags in mobile phones. The following two examples show two real-life application areas that apply RFID:

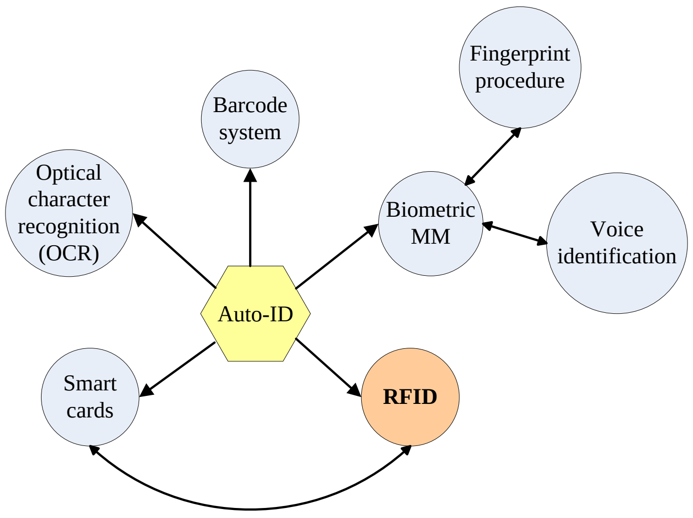


Figure 1.1: Overview of the Auto-ID procedures (adopted from [25]).

METRO Group Initiative The METRO Group Initiative has the goal to modernize the retail industry by deploying RFID. At METRO's Toenisvorst hypermarket, customers can experience the future store, where RFID is deployed. The goal of the project is to test the utilization of the RFID technology (e. g., pilot applications for warehouse management) under real-life conditions.

Pharmaceutical industry The pharmaceutical industry provides a good example for the application of RFID data. H.D. Smith Wholesale Drug Company, a pharmaceutical distributor in the US, is using RFID since the summer of 2005 in order to ensure the pedigree (or authenticity) of the counterfeit-vulnerable pharmaceuticals it distributes [45]. A typical pedigree trail would contain, for instance, a manufacturer, a wholesaler, a chain warehouse and a pharmacy. Since this RFID system aims primarily at eliminating drug counterfeiting and theft, it is crucial for this scenario to have up-to-date information that is easily extractable. At the same time, mechanisms for efficient querying of the drug's pedigree should be provided.

1.1 Problem Statement

Real-world awareness as described by Heinrich in [29] defines the process of extracting real-time information for gaining a better insight into the company's supply chains. Thus, companies are increasingly employing RFID technologies to track their goods in business processes. However, the frequently produced large amounts of events constitute new challenges to modern databases. The main challenges, which are described in detail in Section 2.3, are (1) the huge data volume produced by the RFID sensors, (2) an efficient incremental update (data staging), which must be triggered as soon as new events arrive, and (3) an efficient transactional (OLTP) and analytical (OLAP) query processing. Fast query processing requires up-to-date indexes that are expensive to maintain considering a heavy update load. The challenge of efficient RFID data processing therefore is to manage the trade-off between the required update frequency and a reasonable query response time.

This work concentrates on determining possible data storage and management solutions for RFID data that can cope with the stated challenges. Tracing the movement of individual objects in a sensor infrastructure results in large amounts of data. A medium-sized enterprise, e. g., a single BMW factory, records about 500 events per second (see estimation in Section 2.2). Further, a world-wide operating enterprise, e. g., all BMW factories worldwide, needs to accommodate ca. 2000 events per second. It is of particular importance that efficient query processing is provided in the context of RFID data. OLTP queries should be executed up-to-the-second, since positional updates may occur every second. An appropriate query response time should be achieved for OLAP queries as well. We therefore need an infrastructure that can manage this vast amount of data and can cope with the update frequency as well as with the query rate. Further, traceability information has to be updated continuously, i. e., as soon as new events arrive at the system. This way, query processing is based on up-to-date data. Accessing the most recent events is not only crucial for OLTP applications that are, e. g., interested in locating the last position of an object, but it is also an upcoming requirement for business intelligence applications executing OLAP queries. We explore the capabilities of common relational database systems with respect to managing RFID data. Moreover, we focus on a dedicated approach for storage and processing of traceability data, which is designed to fulfill the requirements of an RFID scenario. Since with the growing amount of event data a centralized approach will reach its limits, we deal with the topic of distributed management of RFID data.

The approaches for management of RFID data can be applied for management of traceability data in general.

1.2 Contributions

There are three main contributions of this thesis:

Relational DBMS Approaches for RFID Management We identify and describe the challenges posed by RFID data. Based on that, we analyze existing approaches that propose different data schemas for efficient RFID management on common DBMS. We implement these solutions and compare them according to the requirements for RFID data. A mixed workload, consisting of concurrent inserts and queries is defined for the approaches in order to measure the realistic achieved throughput. Further, we present a novel approach, where the OLTP and the OLAP components reside in the same relational database, and compare its performance to the existing solutions.

A Dedicated Triple Store for RFID Data A scalable dedicated system for efficient storage and fast querying of RFID data, the RFID Triple Store, is introduced. We design RFID-specific indexes for efficient event processing and use the specific RFID properties to optimize the query engine of the RFID Triple Store. We experimentally evaluate our system by using a mixed workload consisting of inserts (data staging process) and queries (OLTP and OLAP) and analyze the sustained throughput that can be maintained. Further, we compare the performance of our system to the performance of a commercial row-store and a non-commercial column-store database system.

Distributed RFID Management Mechanisms for distributed RFID data management are explored. We evaluate how suitable MapReduce, a technique for large-scale data processing, is in an RFID scenario. Further, we implement a framework for distributed query processing using the RFID Triple Store as a storage system.

1.3 Outline

The remainder of this thesis is organized as follows:

Chapter 2: Characteristics of RFID Traceability Data

This chapter explores the characteristics of RFID data and summarizes the challenges that this data poses to the storage systems. Further, it gives an overview of an RFID event representation and the amount of events that are generated per second in a real life environment.

Chapter 3: Relational DBMS Approaches for the Efficient Management of RFID Data

This chapter presents the main existing work in the field of RFID data management and analyzes the presented approaches with respect to the insert frequency and the query performance. We define a mixed workload, consisting of concurrent inserts and queries and measure the performance of the approaches for this workload. Further, we propose a novel approach for RFID data management, which combines the OLTP and OLAP part in one system and proves a competitive performance for our mixed workload.

Chapter 4: A Dedicated Triple Store for RFID Data Management

This chapter introduces a scalable dedicated solution for efficient management of RFID data. We leverage the characteristics of RFID data to build RFID-aware index structures in order to speed up inserts. Further, we optimize the query engine for the typical traceability queries: range queries over a time interval. Finally, we conduct a comparison between our approach and the current state-of-the-art approaches and show that the RFID Triple Store outperforms its counterparts.

Chapter 5: Distributed RFID Data Management
This chapter investigates mechanisms for distributed RFID data management. We apply the MapReduce technology and analyze whether it is suitable in an RFID context. Further, a distributed query processing using the RFID Triple Store is presented.

Chapter 6: Conclusions and Outlook
This chapter summarizes our findings and concludes this work. It also gives an overview of possible future work topics.

2 Characteristics of RFID Traceability Data

2.1 RFID Event Data

In this section, we provide some background information about the RFID technology. Further, we introduce the triple representation of an RFID event, which is used throughout this work.

2.1.1 RFID Technology

The RFID technology [25] enables the automated tracking of moving objects without line of sight by the use of electromagnetic waves. Data is exchanged between an RFID tag and an RFID reader. The reader usually sends the information to computer systems running RFID software or RFID middleware for further processing. An RFID tag or transponder consists of an antenna and a microchip as shown in Figure 2.1.

RFID tags differ in radio frequency, transfer rate, life time, costs, memory, reading and physical range. There are two main types of RFID tags: active and passive ones. An active tag has an on-board battery and periodically transmits its ID. A passive tag is respectively cheaper and smaller because the battery is absent. It uses the radio energy transmitted by the reader as its energy source. This leads to a shorter reading range since the sensor must be close enough to transfer sufficient power to the tag. We also distinguish between read-only and read/write tags. The first ones use a pre-assigned serial number that is used as a key in a database. The second ones can write and store object-specific data into the tag.

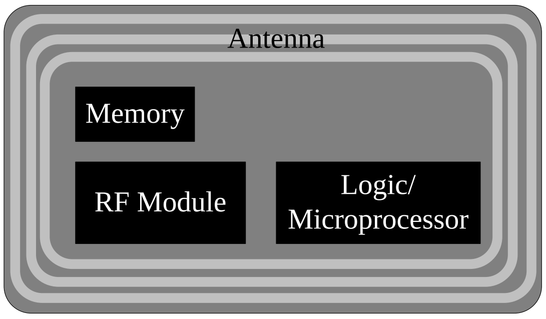


Figure 2.1: A basic RFID Tag. The antenna receives radio signals. Memory size is just a few bytes in order to store an ID number. The memory type can be read-only or read/write. The RF Module uses the antenna to send information back to the reader. The logic unit responds to information sent from the reader. (adopted from [29]).

2.1.2 Electronic Product Code (EPC)

RFID tags carry an Electronic Product Code (EPC) [23], which allows world-wide interoperability. The specification of the RFID tag is published by the EPCGlobal standard [23]. The EPC standard defines a family of coding schemes used to identify manufactured items and to meet the needs of a global RFID usage. EPC tags can identify individual items rather than just a manufacturer or a class of products, in contrast to using barcodes. If a tag is compliant with an EPC standard, its uniqueness is guaranteed.

Currently, EPCGlobal supports EPC codes up to 198 bits. The most commonly used standard, that we also use in this work, however is the SGTIN-96 standard, which defines the EPC length to be 96 bits. In Figure 2.1, the notational conventions for EPC tag encoding schemes are described. A further specification of the different parts of the code (the columns of the table) is given in [23].

2.1.3 RFID Events

Of particular importance is the question how the generated RFID data looks like. Every time when an object is sensed by an RFID reader, an RFID event is generated. An object tracking event can be described by three attributes: the EPC tag *e* that uniquely identifies the object, the reader ID *r* that interrogated the tag carrying the EPC, and the timestamp *t* when the event was generated. RFID events are therefore represented as triples of the form:

$$\textbf{(e, r, t)}$$

EPCs (as described in Section 2.1.2) are character sequences encoding a product's group, producer, and serial number. Typically, the sensor infrastructure is deployed within the processing or supply chain of a company and remains comparatively stable. Objects enter the infrastructure, pass the sensors along the processing

	Header	Filter Value	Partition	Company Prefix	Item Reference	Serial Number
	8	3	3	20–40	24–4	38
SGTIN-96	0011 0000 (Binary value)	(Refer to Table 5 for values)	Refer to Table 6 for values)	999,999 – 999,999,999,999 (Max. decimal range [1])	9,999,999 – 9 (Max. decimal range[1])	274,887,906,943 (Max. decimal value)

[1] Max. decimal value range of Item Reference field varies with the length of the Company Prefix

Table 2.1: Notational conventions for EPC tag encoding schemes (adopted from [23]).

pipeline, eventually exiting the infrastructure after a specific number of processing steps. Depending on the application scenario, the number of generated events per second can be as high as several thousands. The data is generated continuously by the sensor infrastructure and is usually passed to the storage system in batches of a pre-defined size.

Even though the current RFID technology is becoming more stable, readings must still be considered generally inaccurate. The wireless communication, which is used to transfer data between the tag and the reader, is not always reliable due to, e. g., radio-frequency collisions and signal interferences. These technical difficulties lead to tags being missed (so-called false-negative readings) or unexpected extra readings (false-positive readings or "noise"). Duplicate readings can also be often produced if an object stays at the same location for a long time or by tags in overlapping scan areas that are read by multiple readers. In order to eliminate these types of erroneous readings, semantic post-processing, cleaning, and filtering must be performed at a middleware layer before the events are transfered to the managing system. This work, however, does *not* further discuss such issues, but rather concentrates on how the post-processed stream of triples is subsequently stored in a database. Existing work on the subject of data cleaning is briefly discussed in Section 3.4. We assume the data to be complete and correct, i. e., that any necessary cleaning steps have been conducted as a pre-processing step by some of the algorithms described in Section 3.4.

2.2 RFID Application Scenarios

In this chapter, a typical application scenario for the usage of RFID data is presented. Further, we give a realistic estimation of the event generation frequency, i. e., the amount of data that is produced per second. Our benchmarks are based on this estimated value.

BMW

According to the BMW business report from 2005 [8], 1.2 million cars have been produced in 23 production sites. One car consists of about 20000 parts. The estimation of the event generation frequency is based on the following assumptions (adopted from the work of Sosnowski [48]):

- 1000 of the 20000 parts per car are tagged by an RFID chip.
- 20 events per part are generated until they are built into the car.
- 16 working hours per day are assumed.
- 220 working days per year are assumed.

This results in about 1890 RFID events per second for all BMW factories worldwide. We are geared to this estimated value when we design our benchmark experiments. Considering only the factory in Leipzig, up to 650 cars a day are produced. This results in about 226 events per second for a dedicated BMW factory. We, therefore, take the value of 500 events per second as a measure of the average event generation frequency for a small business company and the value of 2000 events per second as the event generation frequency for a world-wide enterprise.

2.3 Challenges Posed by RFID Traceability Data

The specifics of RFID data result in a number of challenges for modern databases. We define and summarize these requirements here. They are used later in this work in form of metrics for the performance of the evaluated storage solutions.

2.3.1 Data Volume

Tracing the movement of each individual object in a sensor infrastructure results in large amounts of data. If we take the estimated event generation frequency of 500 events per second from Section 2.2, and assume a 10 hours working day, we will get about 18 million events per day. A world-wide operating enterprise (e. g., all BMW factories worldwide) needs to accommodate even 2000 events per second. The challenge here is obvious: we need an information management infrastructure that can manage this vast amount of data and can cope with the update frequency as well as with the query rate.

2.3.2 Data Quality

RFID tags work under low-power and low-cost constraints. As already mentioned, wireless communication may not always be reliable due to radio-frequency collisions and signal interferences, metal shielding, or other physical obstructions.

To eliminate these false readings, semantic post-processing, cleaning, and filtering must be performed at a middleware layer. This aspect of RFID data management is beyond the scope of this thesis. The considered storage solutions assume that the generated events are correct and complete, i. e., that data cleaning was performed as a pre-processing step.

2.3.3 Arbitrary Object Movement

In most traceability scenarios, objects move in groups and split into smaller groups. This tree-like object movement has to be mapped on the data model. More complex scenarios, however, come along with more complex, graph-like object movements. That kind of movement can be seen in a post office, where parcels that come from a lot of different small post offices are gathered in one central post office. If a mail is returned to its sender, a cycle occurs in our movement graph. Therefore, we also need storage solutions that can deal with cyclic object movements.

2.3.4 Data Staging

The process of propagating incoming events into a particular data model is defined as *data staging*. In contrast to traditional warehouses, where updates occur only at predicted time intervals, traceability information has to be updated continuously. As soon as new events arrive, the data staging process must be triggered. Accessing the most current information is crucial not only for OLTP applications that are, e.g., interested in locating the last position of an object, but it is also an upcoming requirement for decision processes based on business intelligence applications executing OLAP queries. Therefore, we need an efficient incremental update (data staging).

2.3.5 Query Processing

Fast response times for both OLTP and OLAP queries are also a challenge when managing traceability data. A typical OLTP query in a traceability scenario determines the path of an object (*pedigree query* [5]). A group of common OLAP queries for this scenario are the *contamination queries*: e. g., determine which products have been stored together with product X during a time window; if these products are incompatible, an alert could be produced by the application. Fast query processing requires up-to-date indexes, however high update frequency is crucial in an RFID scenario. The challenge here is to manage the trade-off between the required update frequency and a reasonable query response time.

3 Relational DBMS Approaches for the Efficient Management of RFID Data

This chapter discusses the current existing work for storing and querying RFID traceability data in relational databases. We analyze, implement, and compare the major approaches on RFID data management. However, they either ignore the OLTP part of the data and focus on the OLAP data, or have a hybrid approach, where OLTP and OLAP reside in different systems. We present an innovative database approach for managing traceability data, which merges the OLTP and OLAP components in one system. This solution allows business intelligence applications to consider the "latest" data for their decision processes.

3.1 Existing RFID Approaches on Relational DBMSs

In this section, we present existing approaches for efficient RFID data management. We analyze them qualitatively according to the challenges described in Chapter 2 and quantitatively, measuring their performance in Section 3.3. Further, for the quantitative analysis we consider a mixed workload, consisting of concurrent inserts and queries (OLTP and OLAP). This is very important in an RFID context, since the expected high insert frequency must be managed by the approaches and, at the same time, a reasonable query response time needs to be provided. Mixed workloads, however, were not considered by previous work.

A simple example

As described in Section 2.3, RFID objects may take different movement patterns when traversing a graph. Objects build a *cluster* if they travel at the same time

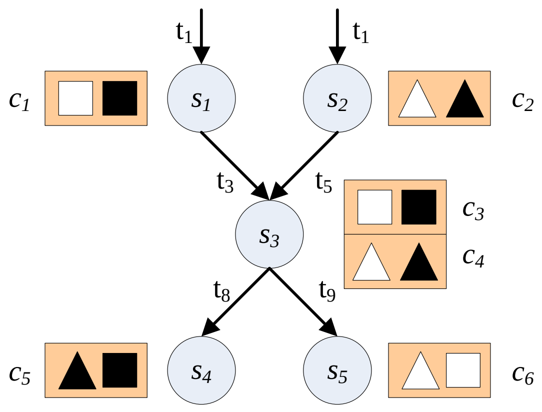


Figure 3.1: Squares and triangles move from sensors s_1 and s_2 to s_3 where they are re-clustered. Black shapes move to s_4, white shapes to s_5. A timestamp at the arrow entering a sensor denotes the arrival time of a cluster.

from one location to another. Figure 3.1 illustrates the movement of four objects through a sensor network consisting of five sensors. The objects in clusters c_1 (objects □ and ■) and c_2 (△ and ▲) move from sensors s_1 and s_2, respectively, to s_3. The objects are re-clustered at sensor s_3 (they build clusters c_3 and c_4), where the ■ object merges with the ▲ and moves to s_4 (cluster c_5). The remaining □ and △ objects move together to s_5 (cluster c_6).

We use the graph in Figure 3.1 as an example for explaining the operating principles of the different database approaches.

3.1.1 Naïve Approach

The naïve approach represents a basic approach and is referred to as a baseline in different works on RFID. It stores each incoming event as a row in table *EVENT*. The table schema and an example are shown in Figures 3.2a and 3.2b.

Data Model

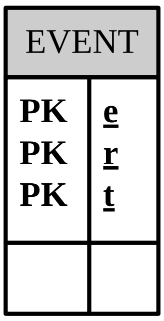
This approach stores all data (consisting of the object's identifier e, the sensor id r which reported the event, and the timestamp t when the event was generated) in one huge table. The model does neither materialize the path of an object nor store information about the belonging of an object to a cluster, i. e., the path or cluster information must be generated at runtime.

Data Staging

The data staging procedure is simply inserting each triple (an RFID event) in the database as shown in Algorithm 1. Data staging is very efficient for this approach,

Algorithm 1: Data staging

```
input: A batch of events tmp of the form: ⟨EPC e, Rdr r, TS t⟩
1 foreach tuple of tmp do
2 |   insert ⟨e, r, t⟩ into table EVENT;
3 end
```



(a) *EVENT* table schema

EVENT		
e	**r**	**t**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_1
...	...	...

(b) *EVENT* table example

Figure 3.2: The naïve approach. Left: the schema of the approach. Right: an example based on Figure 3.1.

since it executes an one-tuple-insert in the *EVENT* table for each event and does not pre-aggregate any information. Therefore, the algorithm is independent of the object's cluster size, i. e., every object is handled independently from the rest and no common cluster information is aggregated.

Query Processing

When we query the naïve model, we have to process the whole *EVENT* table. Therefore, appropriate indexes should be created for reasonable query response times. Because of indexing, the naïve database design supports efficient querying for some query types, e. g., selection of a particular EPC or reader ID. However, queries that compute a relation between two objects or a particular path pattern enforce the use of self-joins, which is extremely costly for the huge table.

We will show in the evaluation section that some of the typical RFID queries are quite time-consuming using the naïve approach.

Advantages and Disadvantages

A clear advantage of the naïve database approach is that it supports an efficient incremental update. This relies on the fact that no information is pre-computed in advance. Working on the "raw", not aggregated information, however, can be disadvantageous for the query processing. We need a considerable set of indexes

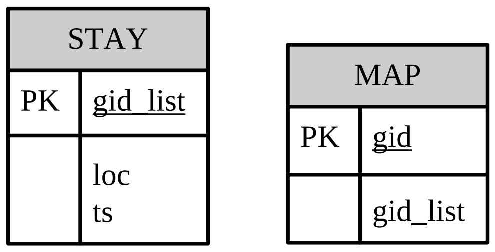


Figure 3.3: Tables for storing events in the warehouse model devised by Gonzalez et al. [27]

for this huge table, which also consume considerable disk space. Further, queries that compute a cluster for a particular object (e. g., which objects travelled together from one reader to another) cannot be implemented efficiently. Since we do not store any type of aggregated history information, the latter has to be computed at query time, which results in a performance decrement. One further disadvantage of this approach is the big disk space overhead, which is explained by the fact that the table grows proportionally with each new event.

3.1.2 Data Warehouse Approach (Gonzalez et al.)

Gonzalez et al. [27] devise a data model that aggregates and compresses the path data of objects based on the observation that objects move in clusters, i. e., groups of objects that move together from one sensor to the next. The approach in [27] assumes that the movement of clusters in, e. g., a retailer scenario, can be visualized as a tree: Products move in large groups and split into smaller groups as they travel from the factory to the distribution centers and then to the stores.

Data Model

In order to store the movement of products, the Gonzalez et al. approach splits the data in two tables: *STAY* and *MAP* (the schemas are shown in Figure 3.3)[1]. The attribute names are depicted as in the original paper – *gid* stands for group ID and loc represents the reader r that scanned the object. Table *MAP* stores the hierarchy of clusters, i. e., how a cluster splits into sub-clusters as it moves through the sensor network. Each row *(gid, gid_list)* represents a parent-child relationship, where *gid_list* contains the list of clusters (identified by *gid*) that stem from the cluster represented by *gid*. The column *gid* contains path-dependent information that encodes the hierarchy of the clusters. Let the string s denote the ID of a cluster whose objects move from the current location to n different locations. When the i-th sub-cluster reaches the new location, we create the cluster ID by concatenating the string ".i" ($0 \leq i < n$) to s and update the *MAP* table: For the first sub-cluster

[1]We omit table *INFO*, which is described in [27]. The table contains path-independent information about products, e. g., the name of the product, manufacturer, and price.

MAP	
gid	**gid_list**
0.0	0.0.0
0.1	0.1.0
0.0.0	0.0.0.0, 0.0.0.1
0.1.0	0.1.0.0, 0.1.0.1
0.0.0.0	■
0.1.0.0	▲
0.0.0.1	□
0.1.0.1	△

(a) Example of table MAP

STAY		
gid_list	**loc**	**ts**
0.0	s_1	t_1
0.1	s_2	t_1
0.0.0	s_3	t_3
0.1.0	s_3	t_5
0.0.0.0, 0.1.0.0	s_4	t_8
0.0.0.1, 0.1.0.1	s_5	t_9

(b) Example of table STAY

Figure 3.4: Warehouse data for data model Gonzalez et al. [27]

($i = 0$), we add a new entry $(s, s.0)$. For all other sub-clusters ($i > 0$), we append $s.i$ to the list of sub-clusters of cluster s. For example, the highlighted row in Figure 3.4a indicates that cluster 0.0.0 split into two clusters, 0.0.0.0 and 0.0.0.1. At the bottom of the cluster hierarchy, *MAP* maps clusters to the list of objects that are contained in the clusters. In our example, clusters 0.0.0.0, 0.0.0.1, 0.1.0.0, and 0.1.0.1 contain only a single product each.

Based on the observation that objects move in clusters, table *STAY* (column *ts*) stores the information when a particular cluster arrived at a location (*loc*). As described above, the cluster identifiers encode how clusters split. *STAY* stores the re-clustering of objects in column *gid_list*, which contains a list of identifiers of clusters that move together from one location to the next. If objects stemming from n different clusters merge into a new cluster, *gid_list* contains a list of n cluster IDs. Using our running example from Figure 3.1, we show the list of cluster IDs in the highlighted row in Figure 3.4. Items ■ and ▲ move together in a cluster from sensor s_3 to s_4. The cluster id 0.0.0.0, 0.1.0.0 indicates that item ■ stems from cluster 0.0.0 (coming from s_1) and ▲ from cluster 0.1.0 (coming from s_2).

In Figure 3.5, we show the cluster concept for the warehouse approach, extending the example in Figure 3.1. In contrast to Figure 3.1, objects are considered to belong to the same cluster not only if they move together from one sensor to another within the same time interval, but if they share the same path from the beginning (from "birth"). At sensor s_3 in Figure 3.5, items □ and ■ belong to the same cluster, because they travelled together from the beginning (path: sensor s_1, sensor s_3). However, objects ■ and ▲ do not belong to the same group at sensor s_4 since their complete path (pedigree) is not the same (■ originates from sensor

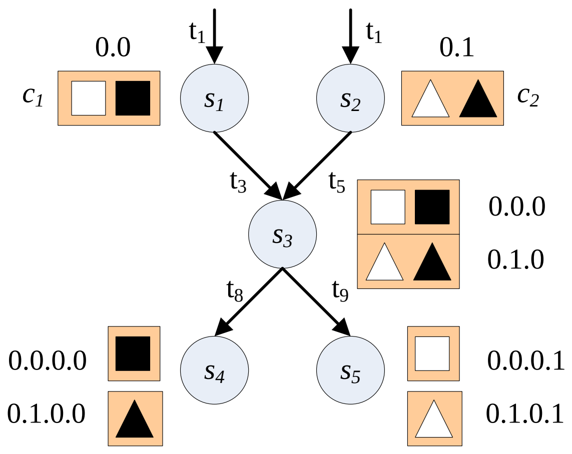


Figure 3.5: The movement graph shown in Figure 3.1 is adapted to the cluster concept of the warehouse approach [27].

s_1 and ▲ from sensor s_2).

Data Staging

The data staging procedure of this approach is not intuitive and has to be explained in detail. The RFID data is aggregated by re-using common object paths. Thus, for every new event it is first checked if the object ID already exists in the database and if so whether there is a path that can be re-used for this object.

The data staging process is depicted in Algorithm 2. A batch of incoming events of the form ⟨EPC *e*, Rdr *r*, TS *t*⟩ is stored in the data structure *tmp*. In order to find out if the entries in *tmp* have already been scanned by a reader and if their path can be continued or whether a new path is needed, we do the following: First, we perform a left outer join with table *MAP*, joining on the columns *tmp.e* and *MAP.gid_list*, which store the EPC of an object. If the join result contains a non-zero *gid_list* entry, the incoming object with EPC *tmp.e* already exists in the database. Second, we perform another left outer join between the result of the first one and table *MAP*, joining the determined *gid* with *MAP.gid_list* in order to find the parent node of the object's *gid* if it exists. This information is needed when updating the existing object's path.

In *inputData*, we store following data stemming from the result of the two left outher joins (line 1): ⟨*e, r, t, gid, parent_gid*⟩, where *e* is the EPC of the incoming event, *r* is the sensor ID that read the EPC, *t* is the current timestamp, *gid* is the generalized ID of the EPC (if it exists), and *parent_gid* stores the parent node of the *gid* value. In Figure 3.4a for example, the *gid* for object ■ has the value 0.0.0.0 as in tuple $\langle 0.0.0.0, ■\rangle$, and its *parent_gid* has the value 0.0.0 because of the entry $\langle 0.0.0; 0.0.0.0, 0.0.0.1\rangle$ that exists in *MAP*.

We process every record of *inputData* and distinguish between three different

Algorithm 2: Data staging of the approach devised by Gonzalez et al. [27] (adopted from [56]).

```
input: A batch of events $tmp$ of the form: $\langle$EPC $e$, Rdr $r$, TS $t\rangle$
1  $inputData \leftarrow (tmp \ltimes_{tmp.e=MAP.gid\_list} MAP) \ltimes_{gid=MAP.gid\_list} MAP$;
   /* inputData contains tuples of the form
      ⟨e, r, t, gid, parent_gid⟩                                    */
2  $new\_parent \leftarrow null$;                    /* the new parent node */
3  $new\_gid \leftarrow null$;                                /* new gid */
4  $child\_count \leftarrow null$;                 /* number of child nodes */
5  $cluster\_gid \leftarrow null$;                       /* common cluster */
6  Sort $inputData$ by $r, t, parent\_gid$;
7  foreach $row \in inputData$ do
8      if $gid = null$ then
9          $cluster\_gid \leftarrow$
           Search in table $STAY$ for an existing cluster path and select its $gid$;
10         if $cluster\_gid \neq null$ then                   /* path exists */
11             $child\_count \leftarrow$ Count the child nodes of $cluster\_id$;
12             if $child\_count = 1$ then
13                 $new\_gid \leftarrow cluster\_gid$ ::'.0';
14                 Update set $MAP.gid = new\_gid$ where $MAP.gid = cluster\_gid$ in table $MAP$;

15                 Insert the row $\langle cluster\_gid, new\_gid\rangle$ in table $MAP$;
16             end
17             $new\_gid \leftarrow cluster\_gid$ ::'.child_count';
18             Insert the row $\langle cluster\_gid, new\_gid\rangle$ in table $MAP$;
19             Insert the row $\langle new\_gid, e\rangle$ in table $MAP$;
20         else
21             $child\_count \leftarrow$ Count the child nodes of $new\_parent$ ID;
22             $new\_gid \leftarrow$ 0'.child_count';
23             Insert the row $\langle new\_gid, e\rangle$ in table $MAP$;
24             Insert the row $\langle new\_gid, r, t\rangle$ in table $STAY$;
25         end
26     else if $parent\_gid = null$ then
27         $new\_gid = gid$::'.0';
28         Insert the row $\langle new\_gid, r, t\rangle$ in table $STAY$;
29         Insert the row $\langle gid, new\_gid\rangle$ in table $MAP$;
30         Update set $MAP.gid = new\_gid$ where $MAP.gid\_list = e$ in table $MAP$;
31     else
32         $new\_parent \leftarrow$
           Search for an entry in table $STAY$ with $loc = r$ and $time\_in = t$ and
           with $STAY.gid\_list$ in $MAP.gid\_list$ where $MAP.gid = parent\_gid$;
33         if $new\_parent = null$ then
34             $new\_parent \leftarrow gid$;
35             $new\_gid \leftarrow gid$ ::'.0'';
36             Insert the row $\langle new\_parent, r, t\rangle$ in table $STAY$;
37         else
38             $child\_count \leftarrow$ count the child nodes of $new\_parent$ in table $MAP$;
39             $new\_gid = gid$ ::'.child_count';
40         end
41         Insert the row $\langle new\_parent, new\_gid\rangle$ in table $MAP$;
42         Update set $MAP.gid = new\_gid$ where $MAP.gid\_list = e$ in table $MAP$;
43     end
44 end
```

cases: (1) the *gid* value is null, i. e., the incoming EPC does not exist (lines 8-25), (2) the *parent_gid* value is null, i. e., there is no parent node and the current path ID is the beginning of the path (lines 26-30), (3) both, *gid* and *parent_gid* values exist thus the path has to be continued (lines 31-43).

For the first case: we first check if the EPC belongs to an existing cluster, in order to re-use its path. For this purpose, we search in table *STAY* for the given location and timestamp (line 9). If an appropriate cluster exists (lines 10-19), we count the child nodes in table *MAP* that belong to that cluster. If only one child node exists, i. e., we found only a tuple of the form ⟨0.0.0.0, ■⟩, then the existing *gid* has to be extended and the current node has to be updated to point to the new ID, which requires changes in the *MAP* table (lines 12-15). At the end, the new EPC is assigned to its newly generated ID in the *MAP* table (lines 17-19). If there is more than one child node in table *MAP* for the given *gid*, we don't need to adjust the existing leave nodes of the form ⟨0.0.0.0, ■⟩, but just create a new ID for the new EPC and update the *MAP* table (lines 17-19). If there is no appropriate cluster for the new object in table *STAY*, we create a new consecutive ID and update both tables, *STAY* and *MAP* (lines 20-25).

For the second case: the incoming EPC exists in the *MAP* table, but no parent ID for its cluster exists. A new cluster ID is created by extending the current cluster ID of the object. The new ID is inserted in table *STAY* and table *MAP*. The *MAP* table is additionally updated such that the EPC is assigned to the new ID.

For the third case: the EPC value, its cluster ID and its parent cluster ID exist. If that holds, we check whether the values of the current tuple *row* are the same as the values from the last iteration (line 32). If so, the objects still belong to the same cluster and they have the same parent ID (lines 33-40). This requires a change only in the *MAP* table, where the current ID is extended and is assigned to the incoming EPC (lines 41-42).

The third case shows the advantage of the warehouse approach. It handles big clusters very efficiently since only small changes in the *MAP* table are required, where the hierarchical identifiers are extended once and the new object IDs are inserted. In the case that no appropriate cluster is found in the *STAY* table, a new record has to be inserted there and afterwards the entries in the *MAP* table have to be adjusted.

Query Processing

The query processing of the approach suffers from the hierarchical identifiers in tables *STAY* and *MAP* that have to be resolved in order to navigate to the EPC of an object or to its parent. Even queries determining the last position of an item, are not able to query only table *STAY*, but have to perform a join between the two tables *STAY* and *MAP* in order to find the EPC. The pedigree of an object is calculated following the hierarchical identifiers in both tables. For more complex queries, like the contamination query, the computation of the cluster IDs yields a considerable overhead. The approach can, however, efficiently calculate the query:

"Which objects were scanned together by reader *r* at timestamp *t*?". In this case we just need to match all cluster IDs in *STAY* that fulfill the conditions to those in *MAP* and select the corresponding EPCs.

Advantages and Disadvantages

The approach described by Gonzalez et al. [27] contains information about the split and merge history of the clusters. This assembling and disassembling feature is an advantage of the database design, since it is an additional information that allows to work on a cluster level rather than on a single object level. Further, this information is not available in other approaches like the naïve approach and if demanded, needs to be computed at runtime. The approach suffers, however, from the database schema, which implies the implementation of the *gid* and *gid_list* attributes as a string. For this reason the data staging procedure performs less efficient than those of the other database models. Almost all queries have to use the recursion, defined in the *MAP* table, in order to resolve the child-parent dependencies and perform therefore worse than for the other database designs. However, queries which use cluster information can be answered efficiently. Another disadvantage of this approach is the extreme disk space consumption, incurred by the string attributes and the indexes created on them.

3.1.3 Read and Bulk Approach (Krompass et al.)

The Read and Bulk (RnB) data model [35] also assumes that objects move in clusters and materializes the path of an object. However, it solves the hierarchy between paths in a more efficient manner than Gonzalez et al. [27]. The RnB approach uses a path definition instead of a cluster definition. A *path* is defined as the sequence of sensors that scanned a particular object. For example, the path of the ■ object in the graph in Figure 3.1 is s_1, s_3, s_4.

Data Model

The schema of the RnB approach is shown in Figure 3.6. Again, we follow the notation of the attributes from the original paper. Table *READ* stores the current (last) location of an object and the timestamp at which it was scanned. It enables to answer OLTP queries concerning the last position of an item efficiently. This table references table *PATH* (column *pid*), which materializes the path of a group of objects that travel along the same path and thus stores historical information about the objects. Table *PATH* is suitable for analytical queries (OLAP) that consider the aggregated historical data. An example of how the object movement is handled in the RnB approach is shown in Figures 3.7a and 3.7b. In order to minimize redundancy, objects that move together share the same path, i. e., reference the same entry in table *PATH*. An entry in *PATH* stores a path identifier *pid* and the time *ts* when the cluster reached the current location *sid*. Each entry is linked to

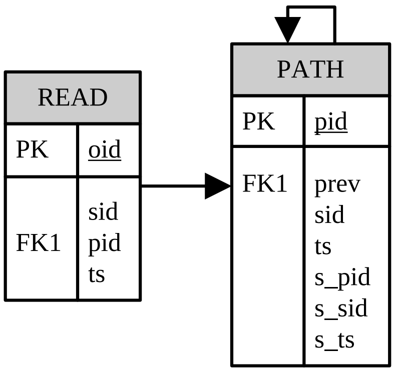


Figure 3.6: Tables for storing events in the RnB data model [35]

the path entry that represents the path to the previous sensor. To facilitate query processing, the path identifiers (*s_pid*), the identifiers of the sensors (*s_sid*) and the timestamps (*s_ts*) when the objects passed the sensors are materialized as a string. The highlighted row in *PATH* in Figure 3.7b shows the path entry with id p_5. Objects referencing this path traveled from s_1 to s_3 and then to s_4 and were last scanned at s_4 at time t_8. The entry references the previous path with id p_3, which represents the movement of an object from s_1 to s_3. Similar to the data model devised by Gonzalez et al. described in the previous section, the RnB data model efficiently stores the paths of objects if large groups of objects split into smaller groups, but objects from different groups do not merge as they move along. Since an entry in *PATH* materializes the entire history of the object movement, n entries must be added to *PATH* if a group contains objects from n different groups. In Figures 3.7a and 3.7b we show how the approach stores the object movement depicted in the graph in Figure 3.1. An example for storing multiple paths for a single group is shown in Figure 3.1 where ■ and ▲ merge at sensor s_3 and move together to s_4. Since ■ and ▲ arrived at s_3 from s_1 and s_2, respectively, two separate path entries in *PATH* are needed to store the movement (p_3 and p_4).

Data Staging

The basic idea of the data staging process of the RnB approach is sketched in [35]. We provide a more detailed algorithmic description of the functionality and explain each step of it.

The data staging process is depicted in Algorithms 3 and 4. A batch of incoming events of the form $\langle$EPC e, Rdr r, TS $t\rangle$ is stored in the data structure *tmp*. First, we perform a left outer join of the table *tmp* with the table *READ* on *EPC*, in order to determine the object IDs of all already existing items in the database (that are present in the incoming batch) and the newly scanned items. Second, the tuples of the result *inputData* are sorted by reader, timestamp, and the path they took so far. Due to the sorting, groups of elements with similar characteristics are created: same reader, same timestamp, and same path. These groups define clusters. Third,

Algorithm 3: Data staging of the RnB approach [35] (adopted from [56])

```
input: A batch of events tmp of the form: ⟨EPC e, Rdr r, TS t⟩
1  inputData ← tmp $\ltimes_{tmp.e=READ.oid}$ READ;
   /* inputData contains tuples of the form ⟨e, r, t, r_last, p⟩ */
2  pioneer ← false;                                  /* a pioneer item? */
3  p_raw_item ← null;                            /* the EPC of the last run */
4  p_raw_reader ← null;                        /* the reader of the last run */
5  p_raw_ts ← null;                               /* the ts of the last run */
6  p_read_path ← null;                    /* the reader path of the last run */
7  new_path_id ← null;                                          /* path ID */
8  insert_action ← false;                              /* insert a new item? */
9  update_action ← false;                              /* update a new item? */
10 Sort the tuples of inputData by $r, t, p$;
11 foreach row ∈ inputData do
12     pioneer ← false;
13     if p_raw_item = null ∨
       ¬(p_raw_reader = row.r ∧ p_raw_ts = row.t ∧ p_read_path = row.p) then
14         pioneer ← true;
15         p_raw_item ← row.e;
16         p_raw_reader ← row.r;
17         p_raw_ts ← row.t;
18         p_read_path ← row.p;
19     end
20     if pioneer = true then
21         if row.p = null;                   /* item was not scanned until now */
22         then
23             new_path_id ← CALL create_new_path_rnb(null, row.r, row.t);
24             insert_action ← true;
25             update_action ← false;
26         end
27         else                                     /* item was already scanned */
28             if row.r = row.r_last then
29                 insert_action ← false;
30                 update_action ← false;
31             end
32             else
33                 new_path_id ← CALL create_new_path_rnb(row.p, row.r, row.t);
                   /* Algorithm 4 */
34                 insert_action ← false;
35                 update_action ← true;
36             end
37         end
38     end
39     if insert_action = true then
40         Insert the row ⟨row.e, row.r, new_path_id, row.t⟩ in table READ;
41     end
42     if update_action = true then
43         Update set READ.sid = row.r, READ.pid = new_path_id,
           and READ.ts = row.t where READ.oid = row.e in table READ;
44     end
45 end
```

READ			
oid	**sid**	**pid**	**ts**
■	s_4	p_5	t_8
□	s_5	p_7	t_9
▲	s_4	p_6	t_8
△	s_5	p_8	t_9

(a) Example of the *READ* table.

PATH						
pid	**prev**	**sid**	**ts**	**s_pid**	**s_sid**	**s_ts**
p_1	$\perp$	s_1	t_1	"p_1"	"s_1"	"t_1"
p_2	$\perp$	s_2	t_1	"p_2"	"s_2"	"t_1"
p_3	p_1	s_3	t_3	"p_1, p_3"	"s_1, s_3"	"t_1, t_3"
p_4	p_2	s_3	t_5	"p_2, p_4"	"s_2, s_3"	"t_1, t_5"
p_5	p_3	s_4	t_{10}	"p_1, p_3, p_5"	"s_1, s_3, s_4"	"t_1, t_3, t_8"
p_6	p_4	s_4	t_{10}	"p_2, p_4, p_6"	"s_2, s_3, s_4"	"t_1, t_5, t_8"
p_7	p_3	s_5	t_{20}	"p_1, p_3, p_7"	"s_1, s_3, s_5"	"t_1, t_3, t_9"
p_8	p_4	s_5	t_{20}	"p_2, p_4, p_8"	"s_2, s_3, s_5"	"t_1, t_5, t_9"

(b) Example of the *PATH* table.

Figure 3.7: Warehouse data for data model RnB [35]

we iterate over *inputData* and determine whether a new cluster begins.

The first element of a cluster is considered as a *pioneer element*. A pioneer element is found if this is the first element inserted in the system or if the sensor ID, the timestamp, or the existing path have changed since the last iteration. In this case the *pioneer* variable is set to *true* (lines 13-14). The current *EPC*, *scanner*, *timestamp*, and *path ID* of the pioneer element are stored for further comparisons with the subsequent elements. Additionally, we check if the pioneer element already exists in the database or if this is the first scan of this object. In the second case, a new path ID is generated using Algorithm 4 and is inserted in table *PATH*. This also means that a new entry containing the new path ID has to be added in table *READ*. Therefore, the variable *insert_action* is set to *true* and the variable *update_action* is set to *false* (lines 24-25). If the pioneer element has already been scanned and exists in the database, we differentiate between two situations: (1) If the sensor ID did not change since the last iteration, there is no action to be done, hence both, *insert_action* and *update_action*, are set to *false*. (2) If the sensor ID changed since the last iteration, a new path ID out of the current path ID is cal-

Algorithm 4: *create_new_path_rnb* (adopted from [56])

```
input  : previous_path (current path ID), reader (current reader ID), ts (current ts)
output: new_id
1  path_id ← null;                                          /* previous path ID */
2  path_reader ← null;                                      /* previous reader path */
3  new_id ← null;                                           /* new path ID */
4  new_path_id ← Look up table PATH for a path with PATH.sid = reader and
   PATH.pid = previous_path and PATH.ts = ts and select its ID;
5  if new_path_id = null then
6      if previous_path not null then
7          path_id ← Get the path history from table PATH;
8          path_reader ← Get the reader history from table PATH;
9      end
10     new_id ← next available ID in table PATH;
11     new_path_id ← path_id :: new_id :: ';';             /* construct new path ID */
12     new_path_reader ← path_reader :: reader :: ';';     /* construct reader path */
13     Insert in PATH row
       ⟨new_id, previous_path, reader, new_path_id, new_path_reader, ts, null⟩;
14 else
15     new_id ← new_path_id;
16 end
17 return new_id;
```

culated using Algorithm 4, hence the variable *update_action* is set to *true* in order to update the entry in table *READ*.

Usually, the pioneer element is followed by other elements belonging to the same cluster. In this case, the algorithm takes advantage of re-using the cluster elements (e. g., the generated path IDs), so that it handles objects belonging to the same cluster efficiently. Further, in contrast to Gonzalez et al. [27], the RnB approach applies numerical identifiers for the path IDs, rather than strings, which yields a better performance for processing them. Until a new pioneer element is encountered, all entries belonging to the same cluster in table *READ* reference the same path ID (determined for the pioneer element) and their sensor IDs are updated.

Algorithm 4 determines the the new path ID of an item. The function expects the following three inputs: (1) the path ID of an item from the last iteration, (2) the current reader ID, (3) the current timestamp value. First, we look up an appropriate path entry for the object in table *READ*, i. e., an object with the same path history, same current sensor ID, and same current timestamp value (line 4). If a suitable path ID is found, it is returned by the function (line 15). Otherwise, a new path entry is generated in table *PATH*. We determine if the current object already exists in the database. If this is the case, we extract the old values from table *PATH*: the path ID's and sensor ID's history (lines 6-8). A new ID is generated (it is assigned the next free numerical identifier) and the values for path ID and sensor ID are updated. The old values are extended by a semicolon and the new path ID and sensor ID is attached respectively (lines 10-12). Finally, the new tuple ⟨*new_id, previous_path, reader, new_path_id, new_path_reader, ts, null*⟩ is inserted in table *PATH* (line 13) and the new ID is returned. This ID can now be referenced by all items belonging to a particular cluster.

Query Processing

This approach is optimized for queries which determine the last sensor that scanned an object, because of the *READ* table containing the current data for an object. It is also efficient if we want to determine the complete path (pedigree) of an element, because this information is explicitly persisted in the schema. It does not need to be computed at runtime as in the approach of Gonzalez et al. [27]. However, if we want to know which objects travelled through some particular stations (i. e., were scanned by particular sensors), we need to extract this information from attribute *s_sid* in Figure 3.7b, which is implemented as a string. The same is true for the timestamp history information.

Advantages and Disadvantages

In contrast to the approach of Gonzalez et al. [27], the RnB approach uses numerical identifiers for the path IDs opposed to the *gid* attribute in [27]). This is beneficial for the staging procedure and the query processing. Because of the numerical identifiers, the approach is less disk space consuming than the naïve and Gonzalez database designs. It reuses one path from the *PATH* table for all objects with identical history. Further, RnB stores explicitly the sensor path history and the timestamp path history per path. Thus, some queries that are interested in determining a path for a particular object do not have to compute it on their own (compare with [27]). A disadvantage of this model, similar to [27], is that clusters can only be split and cannot be merged, e. g., if two objects travelled as a cluster between two sensors and do not have an identical history, they are considered to belong to two different clusters. For this reason, queries concerning the computation of one cluster like, e. g., all objects that were scanned at reader *r* at a particular timestamp, that means queries that operate on the string attributes (*s_pid*, *s_sid*, *s_ts*) perform badly.

3.1.4 Prime Number Approach (Lee and Chung)

The RFID database design of Lee and Chung [37] proposes a sophisticated method for the representation of an object's path using prime numbers. The path encoding scheme and the region numbering scheme (for encoding the time information) used by the approach, apply techniques from research in the XML area [57].

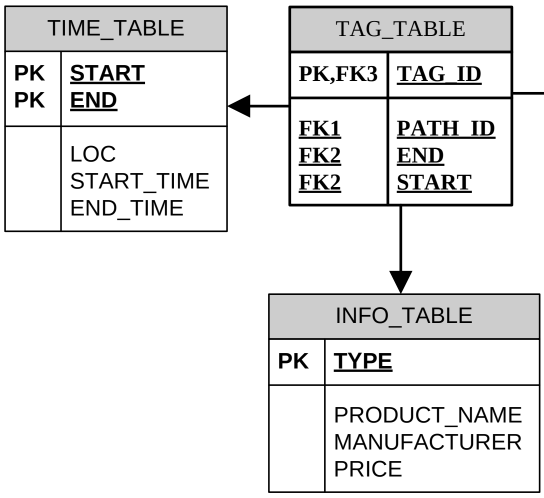


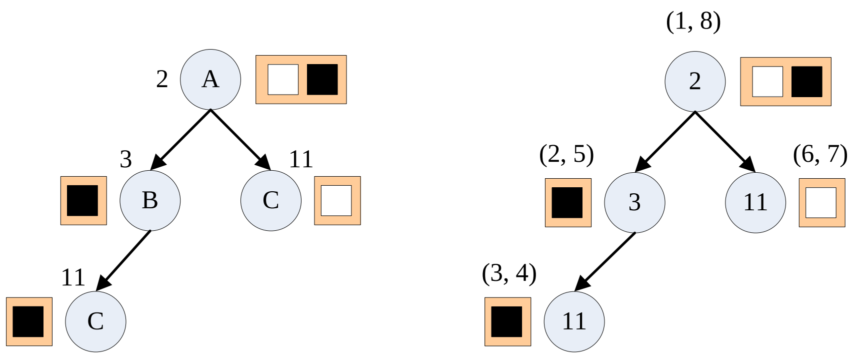
Figure 3.8: Tables for storing events in the prime number approach (adopted from [37]).

Data Model

The relational schema for the approach consists of four different tables as shown in Figure 3.8. *PATH_TABLE*, *TAG_TABLE*, and *TIME_TABLE* are managing the moving of the RFID records, whereas *INFO_TABLE* stores product information like product name, manufacturer, and price. *PATH_TABLE* stores the path information using the prime number encoding scheme, which we explain with the example graph depicted in Figure 3.9. The two columns *ELEMENT_ENC* and *ORDER_ENC* in table *PATH_TABLE* represent the implementation of the path encoding scheme: *ELEMENT_ENC* stands for Element List Encoding Number and *ORDER_ENC* stands for Order Encoding Number. To encode a path, the authors assign a prime number to each location and calculate the product of all prime numbers which occur in the object's path. This product is stored in the attribute *ELEMENT_ENC* in the *PATH_TABLE*. Before we can explain *ORDER_ENC*, we need some more background information.

Figure 3.9a shows that the prime numbers 2, 3, and 11 are assigned to the locations *A*, *B*, and *C*. As we see, the same location can be included in different paths of the graph: location *C*. However, cycles in a path are not allowed, i. e., the same location cannot occur more than once in a particular path. This is due to the mathematical specifics of the approach as we will discuss later. The value of the attribute *ELEMENT_ENC* for the path from location *A* to *C* (2 –> 11) is 22.

Suppose we know the *ELEMENT_ENC* value and want to determine the nodes (locations) which participated in a particular path. Here, the authors make use of the *Fundamental Theorem of Arithmetic* [46], which states that any natural number (except the number 1) is uniquely expressed by the product of prime numbers. Therefore, the value of *ELEMENT_ENC* can be uniquely defactorized in prime



(a) Path encoding scheme. **(b)** Time tree structure.

Figure 3.9: An example of the path encoding scheme and region numbering scheme (time tree) of the prime number approach [37]

numbers and that results in the locations an object passed, i. e., its pedigree.

It is not only important to know at which locations an object was scanned, but also to determine the order in which it traversed the path. This can be done by applying the *Chinese Remainder Theorem* (CRT) [46]. CRT states: Suppose that n_1, $n_2, \ldots, n_k$ are positive integers which are pairwise coprime (i. e., pairwise relatively prime numbers). Then, if a_1, $a_2 \ldots, a_k$ is any given sequence of integers, there exists X between 0 and N $(= n_1 * n_2 * \ldots * n_k)$ solving the system of simultaneous congruences.

$$\begin{aligned} X\ mod\ n_1 &= a_1 \\ X\ mod\ n_2 &= a_2 \\ &\ldots \\ X\ mod\ n_k &= a_k \end{aligned}$$

Knowing this, we substitute the values $n_1, n_2, \ldots, n_k$ with the nodes' prime numbers and the values $a_1, a_2, \ldots, a_k$ with the ordering of the nodes, e. g., 1, 2, etc. Since $n_1, n_2, \ldots, n_k$ are prime numbers, they are pairwise relatively prime. According to the Chinese Remainder Theorem, there exists a number X between 0 and the product of $n_1, n_2, \ldots, n_k$ solving the linear system. This number is stored in the column *ORDER_ENC* in *PATH_TABLE*. For our example in Figure 3.9a, we can determine the value *ORDER_ENC* by solving the linear congruences:

$$\begin{aligned} X\ mod\ 2 &= 1 \\ X\ mod\ 11 &= 2 \end{aligned}$$

Using the Extended Euclidean Algorithm [46], we calculate the value 13 for X (*ORDER_ENC*). Given the Order Encoding Number X, one can determine the order information for any location on the path by solving $X\ mod\ n$, where n is the prime number denoting the node.

In order to store the time information for products, the authors construct a time tree and apply a region numbering scheme as shown in Figure 3.9b. The *TIME_TABLE* contains information about the first scan of an item at a particular location and the time when it leaves this location. Both attributes *START* and *END* allow for an efficient search of the predecessor or successor of a node. Note, that the *START* and *END* columns do not store real timestamp values, but just represent a topologic order of the time information. As depicted in Figure 3.9b, the time tree is built out of the RFID events. Each node represents a location containing time information, the point of time when the item enters the location (start) and the point of time when the item leaves the location (end). If an item was at the same location at two different points in time, then two different nodes in the time tree are needed (location *C* is an example). Further, in order to determine the values for *START* and *END*, a depth-first search is conducted. Here, for each node first the *START* value is assigned and then the *END* value is derived according the depth-first search. Therefore, the region numbering scheme has the property that if a node *A* is a predecessor of node *B*, *A.START* < *B.START* and *A.END* > *B.END*. For instance, consider the item □ traveling from location 2 to location 11 in Figure 3.9b. If we want to determine at what time □ visited the different locations, we look up the last location of object □ in the *TIME_TABLE* and then determine the predecessor of the last location node, i. e., we search for all entries in the table that fulfill the conditions *START* < 6 and *END* > 7. The result is node 2 with *START* $= 1$ and *END* $= 8$.

Data Staging

The encoding scheme can handle tree-like and graph-like object movements (though not very efficiently), but has the drawback that it cannot handle cycles in the object movement, because of the mathematical specifics of the approach. We address these limitations in the following.

Suppose that objects travel through a graph with two or more different start nodes. Applying the Lee and Chung approach results in a contradiction with the region numbering scheme (shown in Figure 3.9b) used for constructing the time tree. If we assign the same topologic order information to both start nodes, the start/end condition of the time tree will be violated. The only possibility to apply the Lee and Chung [37] approach to a graph-like movement is to duplicate the graph except for the start nodes and to handle two tree-like movements. This case is shown in Figure 3.10. We modified the example graph in Figure 3.1 in a way that it is conform with the requirements of the approach.

Further, cycles within a path are not allowed in the object movement and cannot be handled because of the mathematical specifics. The value of the attribute *ORDER_ENC* denotes the ordering of the nodes in the graph. Suppose that 3 nodes with prime number labels 5, 7, and 11 build a cycle in one path of the tree. A valid

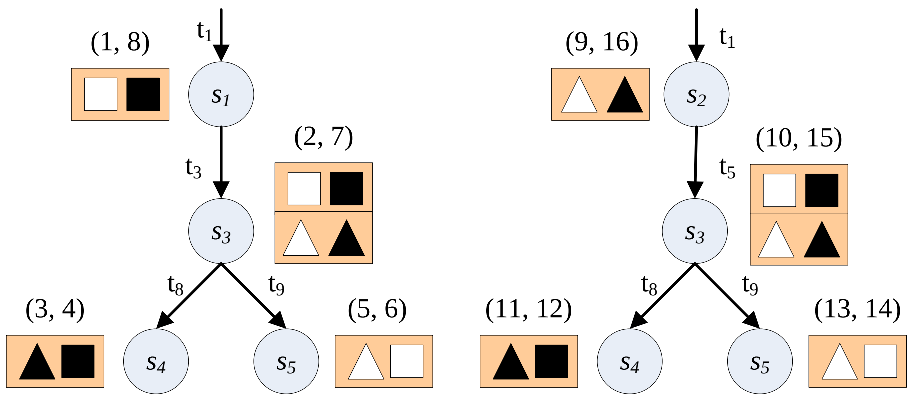


Figure 3.10: Graph-like movement handling in the Lee and Chung approach [37] (based on the example in Figure 3.1).

value X for *ORDER_ENC* should fulfill the following conditions:

$$\begin{aligned} X \bmod 5 &= 1 \\ X \bmod 7 &= 2 \\ X \bmod 11 &= 3 \\ X \bmod 5 &= 4 \end{aligned}$$

According to the Chinese Remainder Theorem [46], however, the congruences can only be solved if the prime numbers are pairwise coprime, which does not hold in this case. Therefore, this approach cannot be used if cycles occur in the graph.

During data staging, we construct the time tree according to the region numbering scheme. If a new batch of events enters the system and some of the items of the new batch have already occurred in the last batch, the affected path of the tree has to be extended by the new location (or by the same location with new time information). The entries in the *TIME_TABLE* have to be updated accordingly. This requires to reconstruct the complete time tree and update all node entries because of the numbering scheme. Therefore, data staging can only be applied if the complete movement of every object is known in advance.

Query Processing

Because of its mathematical background, the approach is particular efficient for path oriented retrieval queries. These queries determine ancestor-descendant relationships between locations, e. g., which objects travelled through location A first and then through location C. Due to the compact representation of the paths, *PATH_TABLE* will be relatively small, but if the prime numbers get large, there will be an overhead to defactorize them. If we are interested in the last position of an object, we have to join *TAG_TABLE* and *TIME_TABLE*, which could grow very big. Some numbers on the query performance can be found in [37].

Advantages and Disadvantages

An advantage of the Lee and Chung approach is the very compact representation of the object's path. This results in a good disk space utilization and efficient query performance for path queries. Some design decisions of this approach (like the described region numbering scheme), however, impose a limitation, because it is not possible to incrementally update the database design unless the complete movement of an object is known in advance. Otherwise, the complete time tree must be reconstructed and the entries in *TIME_TABLE* must be updated. This is extremely time consuming and makes high frequently event updates practically impossible. This means that data staging cannot be performed in the way we require it for the other approaches. Therefore, we omit this approach in our evaluation.

3.2 A Combined OLTP and OLAP Approach for Traceability Data

The approaches presented so far, either ignore the OLTP part of the data and focus on the OLAP data, or have a hybrid approach, where OLTP and OLAP reside in different systems. The naïve model does not distinguish between an OLTP and OLAP part, but focuses only on the OLAP part. The approach of Gonzalez et al. represents a typical warehouse approach; the RnB approach considers a main memory component for answering the OLTP queries.

Real-time business intelligence applications are not only interested in ´´old" data for their decision-making processes, but need to involve the latest information as well. Hasso Plattner discusses the need for a common database approach for OLTP and OLAP since this "could make both components more valuable to their users" [43]. As the trend goes towards combining the OLTP and the OLAP part in one system, we present an approach that is designed to fulfill this requirement. Our approach is based on a new path encoding which enables us to efficiently materialize the movement history of an object and also functions as a sort of index. The data model of the naïve approach is extended so that the most current data is kept separate from the historical data without aggregating or discarding any information. In order to provide an efficient data staging, we pursue an append-only approach, i. e., there exist only inserts and no updates, and consolidate the database regularly. Parts of this work were published in [20].

3.2.1 Path Encoding Using a Bloom Filter

A challenge in a traceability scenario is to find a way to efficiently store the path of an object. Objects "flow" from one sensor to another and their movement history is of interest for traceability data applications. Materializing the path of each object requires a complex pre-processing, i. e., the data staging process gets more complicated. Therefore, various techniques for efficient storage of traceability data and, in particular, the data path (in the example of RFID) have been proposed, see Section 3.1. We propose a Bloom filter solution for path encoding in a traceability scenario.

The Bloom filter is a space-efficient data structure that is used to test whether an element is a member of a set. An empty Bloom filter is a bit array with m bits, all set to 0. To add an element one has to find the corresponding position in the Bloom filter using a defined hash function and set this bit to 1. For a traceability data scenario, we assume that sensors are grouped in geographical regions. The sensor infrastructure is known in advance. New sensors can be mapped to the existing geographical regions. Therefore, we use one bit for every geographical region, not for a sensor, in order to keep the length of the Bloom filter small. When an object was scanned by a reader from a particular geographical region we set this bit to 1. Thus, an object has visited all geographical regions for which the corresponding bit in the Bloom filter is set. Note, that for our approach no false positives can occur, since the length of the Bloom filter matches exactly the number of possible geographical regions and the hash function performs a one-to-one assignment. The Bloom filter is easily extendable in case new regions are added to the application.

3.2.2 Data Model

We aim at creating a database schema that fulfills the requirements of traceability applications. Those applications have to handle millions of events per second while still being able to query the high amount of produced data in order to trace, e. g., lost packets. We design an approach that combines the OLTP and OLAP paradigms in one database to provide the latest information to decision-making business intelligence applications. Our approach exploits the main advantage of the naïve approach, namely its simplicity, while trying to avoid its drawbacks consisting in the fast growth of the table, which results in inefficient query processing. For this reason, we partition the *EVENT* table of the naïve approach into smaller tables, where the readers are organized in regions. A region is a geographical unit that comprises the sensors located in it. The granularity of a region depends on the use case: a region can represent a country, a city, or a single factory.

Figure 3.11 shows our database design. The schema consists of the OLTP table, where the most current data is kept and the REGION tables, where (historical) path information is stored. In the OLTP table, the last occurrence of an object (identified by its *oid*), the sensor that scanned it (*rdr*), and the timestamp (*ts*) when the object passed the sensor are stored. The encoding with the Bloom filter described in Section 3.2.1 is also materialized in the OLTP table. One can determine in which geographical region an object was scanned by determining the positions r_i in the Bloom filter where an 1 occurs. The r_i values specify the corresponding *REGION* tables that hold the information about those (potentially outdated) read operations. Each *REGION* table has the schema of the naïve approach. It stores the objects that were scanned by sensors from the respective region. This means

OLTP			
oid	**rdr**	**ts**	**bloom**
□	s_4	t_n	101
■	s_4	t_n	101
○	s_3	t_1	010
...	...	...	...

(a) Example of the OLTP table.

Region1		
oid	**rdr**	**ts**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_2
...	...	...

(b) Example Region1 table.

Region2		
oid	**rdr**	**ts**
○	s_3	t_1
◇	s_3	t_1
▲	s_4	t_2
...	...	...

(c) Example Region2 table.

Region3		
oid	**rdr**	**ts**
□	s_4	t_n
■	s_4	t_n
△	s_5	t_n
...	...	...

(d) Example Region3 table.

Figure 3.11: Bloom filter approach

that the same object can occur several times in a *REGION* table. After determining in which region an object was (using the *OLTP* table), one can extract from the *REGION* table the explicit information about which sensor (*rdr*) read the object and at what time (*ts*).

This approach efficiently answers both OLTP- and OLAP-style queries. As the name indicates, the *OLTP* table serves OLTP requests which require up-to-date information. The typical OLTP request in a traceability scenario is to determine the last position of one item. OLAP queries, e. g., asking for all readers that one item has passed, can be answered by joining the *OLTP* table and the *REGION* tables to which the Bloom filter points. Queries examining a data flow in only one region only read data from the corresponding region table which is advantageous in query processing.

3.2.3 Data Staging

An efficient incremental update is one of the biggest challenges when designing a data model for traceability data. In this section, we describe how we realize data staging in our approach.

When an item is first read, one tuple is inserted into the *OLTP* table with the Bloom value being 0 at all positions except for the region the item was read in, which is set to 1. A second tuple is inserted into the corresponding region table to which the Bloom filter is pointing. When the same item is read a second time, its reader (*rdr*), timestamp (*ts*), and Bloom filter values in the *OLTP* table are updated, and a tuple is added to the *REGION* table the reader belongs to. Consider the item with *oid* □ in Figure 3.11. It has been scanned twice, at reader s_1 in region 1 and at reader s_4 in region 3. The item with *oid* ■ is in the same cluster as □ and thus passed the same readers at the same time and has the same Bloom filter. The entries in the *OLTP* table for these items contain the latest information: the last time the items were scanned is t_n and the corresponding reader is s_4. The Bloom filter values contain two occurrences of 1 at the positions 1 and 3, i. e., the items □ and ■ have moved through the regions 1 and 3. In contrast, the item with *oid* ◯ has only been scanned once (yet) at reader s_3 in region 2, so that the information in the *OLTP* table and the corresponding *REGION* table is identical and the Bloom filter only points to region 2.

How the data staging algorithm manipulates the tables of the database design is pictured in Figure 3.12. In order to provide an efficient incremental update, we do not insert each single event, but process a batch of events. The more events a batch contains the higher the throughput. However, a database supporting OLTP has to contain the most current data, so there is a trade-off between batch size and data update latency. We consider a batch size of 5000 events to be a reasonable trade-off for our scenario. When regarding applications with very fluctuating event arrival frequencies, we recommend to adjust the batch size to the current arrival frequencies of events. When the frequency is low, it is not acceptable to wait several minutes before executing a batch as this affects the timeliness of the data in the database system. On the other side, during peaks, the high efficiency of batch processing can be exploited to achieve a very high insert throughput.

Since updates are more expensive than inserts, we design our approach to be append-only. Thus, we replace the updates in the *OLTP* table by inserts into the *OLTP* table. We use two auxiliary tables in oder to make use of the efficient batch processing and good insert performance of database systems. The temporary *TEMP* table holds each batch before it is processed. The *Deathlist* table contains outdated events. Figure 3.12 illustrates how a logical update is substituted by two inserts. In the example, items are processed with a batch size of 3. When an item is first read, the processing is as described above (e. g., see the first batch consisting of the tuples with the *oid* values □, ■, and △ in *OLTP* and *REGION1* tables in Figure 3.12a). The second batch consists of the same items, now scanned by different sensors in region 3. When the items are read for the second time, i. e., an update occurs, the corresponding tuples in the *OLTP* table are not updated, but they are copied to the *Deathlist* table, which denotes that these tuples are outdated and represent historical information. In Figure 3.12b, the first three *OLTP* tuples (constituting the first batch in Figure 3.12a) are now also in the *Deathlist*. As the *REGION* tables are not affected by updates, such historical data is still accessible. The latest information is stored by appending an updated tuple (with new reader, timestamp, and Bloom values) to the *OLTP* table (the last three *OLTP* tuples in

OLTP			
oid	**rdr**	**ts**	**bloom**
□	s_1	t_1	100
■	s_1	t_1	100
△	s_2	t_1	100

Deathlist		
oid	**rdr**	**ts**

Region1		
oid	**rdr**	**ts**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_2

(a) After first batch

OLTP			
oid	**rdr**	**ts**	**bloom**
□	s_1	t_1	100
■	s_1	t_1	100
△	s_2	t_1	100
□	s_4	t_n	101
■	s_4	t_n	101
△	s_5	t_n	101

Deathlist		
oid	**rdr**	**ts**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_2

Region1		
oid	**rdr**	**ts**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_2

Region3		
oid	**rdr**	**ts**
□	s_4	t_n
■	s_4	t_n
△	s_5	t_n

(b) After second batch

OLTP			
oid	**rdr**	**ts**	**bloom**
□	s_4	t_n	101
■	s_4	t_n	101
△	s_5	t_n	101

Deathlist		
oid	**rdr**	**ts**

Region1		
oid	**rdr**	**ts**
□	s_1	t_1
■	s_1	t_1
△	s_2	t_2

Region3		
oid	**rdr**	**ts**
□	s_4	t_n
■	s_4	t_n
△	s_5	t_n

(c) After consolidation

Figure 3.12: Data staging of the Bloom filter approach

Figure 3.12b constitute the second batch). The most current data is now computed as the set difference of the *OLTP* table and the *Deathlist* (as shown in Figure 3.12c).

The implementation of batch inserts (and updates) is described in Algorithm 5. Here, we exploit the DBMS's BULK INSERT command and efficient join computation. In lines 1 to 5 the temporary table *TEMP* (referred to as *T*) is created in the database, all tuples to be processed are written to a file *F* and the data contained in *F* is loaded to *T* by executing the BULK INSERT command. In lines 6 to 10, all tuples in the processed batch which represent already outdated data are identified and written to a file *D*, denoting the content of the *Deathlist*. If multiple subsequent reads of the same item are executed in the same batch, only the last read (with the most current timestamp) is valid OLTP data, the rest is historical data. In

Algorithm 5: Algorithm processBatch

```
input : A batch of events $S$ of the form $\langle$EPC $e$, Rdr $r$, TS $t\rangle$

1  create a temporary table $T(e, r, t)$;
2  forall the events $s \in S$ do
3      rewrite $s$ as insert $i$ and append $i$ to file $F$;
4  end
5  BULK INSERT data from $F$ to $T$;
6  forall the EPC values $e \in T$ do
       /* outdated tuples */
7      forall the tuples $d$: $d.epc = e.epc \wedge d.timestamp < e.timestamp$ do
8          write $d$ to deathlist file $D$;
9      end
10 end
11 $U \leftarrow T \bowtie$ OLTP;                /* tuples to be updated */
12 forall the tuples $u \in U$ do
13     write $u$ to deathlist file $D$;
14     forall the tuples $t \in T$ where $t.epc = u.epc$ do
15         write $t$ to OLTP file $O$ with updated Bloom value;
16         write $t$ to corresponding region file $R_i$;
17     end
18 end
   /* tuples to be inserted */
19 forall the tuples $i \in T \wedge i \notin U$ do
20     write $t$ to OLTP file $O$ with Bloom value pointing to the new region;
21     write $t$ to corresponding region file $R_i$;
22 end
23 forall the files $R_i$ do
24     BULK INSERT data from $R_i$ to corresponding region table;
25 end
26 BULK INSERT data from $O$ to OLTP table;
27 BULK INSERT data from $D$ to Deathlist;
```

line 11, the temporary table T is joined with the OLTP table, thereby determining the logical updates within the current batch. All tuples within the join result have to be treated as updates (lines 12 to 18), while the rest of the batch tuples represent "real" inserts (lines 19 to 21). All tuples that are to be updated are written to the deathlist file. Finally, the *OLTP* table, the *REGION* tables, and the *Deathlist* are loaded using the BULK INSERT command.

As described above, OLTP queries require the most current data, which is obtained by computing the set difference between the *OLTP* table and the *Deathlist*. In order to keep the overhead as small as possible and to avoid very large tables,

Algorithm 6: Algorithm consolidate

```
1 $O \leftarrow OLTP - Deathlist$;
2 foreach tuple $o \in O$ do
3     write $o$ to file $F$;
4 end
5 drop OLTP table;
6 drop Deathlist table;
7 create OLTP table;
8 BULK INSERT data from $F$ to OLTP;
9 create Deathlist;
```

we consolidate the *OLTP* table and the *Deathlist* from time to time (i. e., after a certain number of batch inserts). The consolidation procedure (which equals a delete from the *OLTP* table) is described in Algorithm 6. After determining the most current data in line 1 and writing those tuples to a file *F*, we drop both the *OLTP* table and the *Deathlist*, recreate them, and use BULK INSERT to load the data from *F* into the newly created *OLTP* table. The *Deathlist* remains empty. Figure 3.12c illustrates the consolidation. The outdated tuples, which occur in both the *OLTP* table and the *Deathlist*, are removed from the *OLTP* table so that only the latest data is retained. After that, the *Deathlist* is emptied. The result of query processing is now the same as before the consolidation, however it is computed more efficiently. The *REGION* tables are not affected by the consolidation procedure.

3.3 Performance Evaluation and Comparison

In this section, we present the evaluation of the different approaches. Before we show performance numbers, we compare the approaches according to the requirements that should be fulfilled by an RFID database design. Some of the findings in this section were published in [56] and [54] which were supervised by the author of this thesis.

3.3.1 Qualitative Evaluation

Some of the RFID requirements are derived from the challenges discussed in Section 2.3 and some originate from the design of the approaches. We give a short overview of how the requirements are defined and to what extent they are fulfilled. The qualitative comparison is shown in Table 3.1.

Data Staging

As already mentioned in Section 2.3, supporting an efficient incremental update is essential for RFID applications, in order to achieve real world awareness. We classify the approaches according to whether they allow for an efficient data staging. We assume that if new incoming events require that the data staging procedure refactures a big part of the already stored data, then this approach does not support an efficient incremental update. This applies for the prime number approach [37] as stated in Table 3.1. The reasons for that are discussed in Section 3.1.4.

The efficiency of the data staging procedure is determined in the evaluation. Database solutions which do not pre-aggregate the data have a more efficient data staging process.

Handling trees

In most RFID scenarios, objects move in large groups and split into smaller ones. This simplest movement is defined as tree-like movement and is supported by all presented approaches as can be seen in Table 3.1.

Handling DAGs

In some scenarios, more complex object movements are needed. For instance, if we take the post office infrastructure as an example and consider the post offices as sensors, there are parcels that come from a lot of different small post offices and are gathered in one central post office. This constitutes a "merge" of different object groups. This kind of splitting and merging may occur multiple times during the parcels' lifetimes. Thus, we need an implementation of a re-grouping of objects for this scenario, which implies the use of a graph structure, a directed acyclic graph (DAG), instead of a tree. Whether the observed approaches fulfill this requirement is noted in Table 3.1. The naïve and Bloom filter approaches do not distinguish between different object movements, but process each single event independently. Thus, they implicitly allow for handling DAGs. For Gonzales et al. [27] and RnB [35], storing object movements in a graph results in additional overhead, because clusters are considered to have the same path from birth on. Thus, entries have to be stored redundantly if they have different origins, e. g., p_3 and p_4 from table *PATH* in Figure 3.6. The approach of Chung and Lee [37] does not provide a possibility to store a graph movement without modifying the graph as explained in Section 3.1.4.

Handling DCGs

Consider the post office scenario explained above. If a mail is returned to its sender, a cycle occurs in our movement graph. Therefore, we need storage solutions that can deal with cyclic object movements: directed cyclic graph (DCG).

Requirements	**Naïve approach**	**Gonzales et al. [27]**	**Read and Bulk [35]**	**Prime number approach [37]**	**Bloom filter approach**
Data Staging	Yes	Yes	Yes	Not possible efficiently	Yes
Handling trees	Yes	Yes	Yes	Yes	Yes
Handling DAGs	Yes	Yes, but redundantly	Yes, but redundantly	No	Yes
Handling DCGs	Yes	Yes, but redundantly	Yes, but redundantly	No	Yes
Paths of different length	Yes	Yes	Yes	Yes	Yes
Inserting new nodes on demand	Yes	Yes	Yes	No	Yes
Focus	single event	cluster	cluster	cluster	single event

Table 3.1: Qualitative comparison of the approaches with regard to the RFID data management requirements

Again, the naïve and Bloom filter approaches do not distinguish between different object movements and therefore support handling DCGs. Gonzales et al. [27] and RnB [35] store information redundantly if the objects move in cycles. The reasons are the same as these for handling DAGs. The prime number approach [37] does not support cyclic movements because of its mathematical specifics, especially because of the Chinese Remainder Theorem [46], explained in Section 3.1.4.

Paths of different length

Some objects may "stay" at a location and not move any further. This implies that paths of different length occur in the movement tree or graph. All approaches fulfill the requirement of modeling paths of different length, as noted in Table 3.1.

Inserting new nodes on demand

In real-life traceability applications, the sensor landscape can change over time. Additional sensors may be dynamically inserted. A flexible solution for traceability data should be able to cope with such changes. Table 3.1 shows that all approaches except for the Lee and Chung approach [37] fulfill this requirement. The reason why the prime number approach cannot cope with new nodes is the applied time tree, which has to be re-built for each newly inserted node as described in Section 3.1.4.

Focus

The different approaches process the RFID information in a different way. While the naïve and Bloom filter approaches focus on each single event and do not organize the objects in groups, the Gonzales et al., the RnB, and the Lee and Chung approach are interested in clusters and reuse common paths of objects.

3.3.2 Framework Architecture

Figure 3.13 shows a high-level architectural overview of the framework used for evaluating the different RFID approaches. *Sensors* scan objects and send the object's identifier (*e*) along with an identifier of the sensor (*r*) and the time when the object was scanned (*t*) as an event *(e,r,t)* to the *Middleware*. The middleware reads the events and triggers data staging which updates the data in the *Warehouse* incrementally. Events are processed in batches by the staging procedure. As we motivated in Section 2.2, our system has to be able to handle an average data arrival frequency of 500 events per second. This corresponds to the event generation frequency in the production process of a medium-sized enterprise. In this work, we focus on the data staging component and on the different possibilities to store the data in the warehouse. The subject of data cleaning, which is performed by the *Middleware*, is described in Section 3.4 and is beyond the scope of this thesis.

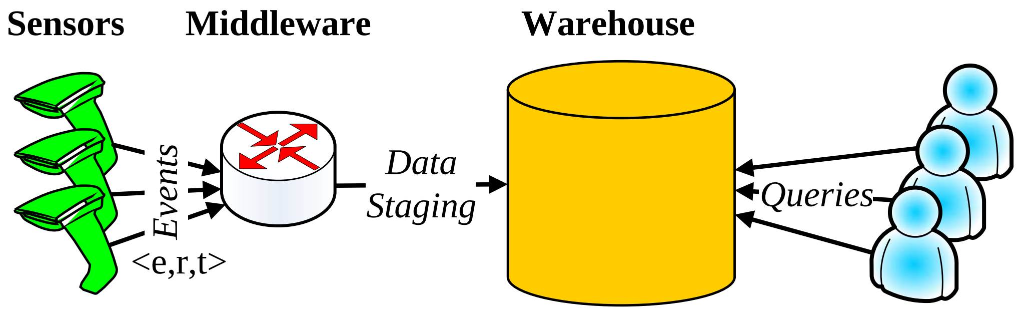


Figure 3.13: Architecture of the framework for evaluating the RFID approaches.

The algorithm that performs data staging is dependent from the data model that is used to store the data in the warehouse.

3.3.3 Evaluating the Existing RFID Database Approaches

In this section, we present a performance comparison between the three approaches – naïve, Gonzalez et al. [27], and RnB [35]. We conducted three sets of experiments to evaluate the different data models: we evaluated the maximum insert throughput that can be achieved, the query-only performance, and the performance of a mixed workload consisting of concurrent inserts and queries.

The experiments were executed on a dedicated host, equipped with two Intel Xeon 3.20 GHz CPUs with 2 MB cache memory respectively, 8 GB main memory and 6 U320-SCSI hard disks, running an enterprise-grade 64 bit-Linux, which serves as a database server. A commercial row-store DBMS runs on the server.

We implemented the three data models except the model by Gonzalez et al. using the table structures as shown in Section 3.1. The logical schema of the Gonzalez et al. design implies the use of composite attributes as for example a list like *gid* in *MAP* and *gid_list* in *STAY*. This is a violation of the first normal form which requires that all attributes have an atomic domain. Therefore, we chose to normalize the tables for this model. Instead of storing lists of values in a row, we duplicate the row for each item in the list. For example, we split an entry *(0.0.0; 0.0.0.0,0.0.0.1)* in *MAP* (the highlighted row in Figure 3.4a) into two rows *(0.0.0; 0.0.0.0)* and *(0.0.0; 0.0.0.1)*.

Table 3.2 summarizes the data types we used in our implementation. The electronic product code (EPC) standard [23] specifies different EPC variants, from which we chose SGTIN-96, i. e., a 96 bit identifier that encodes the manufacturer, the product type, and the serial number of the item (as described in Section 2). Since the database does not provide a dedicated data type for storing EPC values, we encode the identifier as a `VARCHAR` value, in order to have a generic representation that allows the code to contain characters as well. We chose to store the identifier of a sensor as `INTEGER`, since the number of readers in the sensor infrastructure is not supposed to overflow this value range. At this point, we do

	Column	Data Type
Naïve	EVENT.oid	`VARCHAR`
	EVENT.sid	`INTEGER`
	EVENT.ts	`TIMESTAMP`
Gonzalez et al.	MAP.gid	`VARCHAR`
	MAP.gid_list	`VARCHAR`
	STAY.gid_list	`VARCHAR`
	STAY.loc	`INTEGER`
	STAY.ts	`TIMESTAMP`
RnB	READ.oid	`VARCHAR`
	READ.sid	`INTEGER`
	READ.pid	`INTEGER`
	READ.ts	`TIMESTAMP`
	PATH.pid	`INTEGER`
	PATH.prev	`INTEGER`
	PATH.sid	`INTEGER`
	PATH.ts	`TIMESTAMP`
	PATH.s_pid	`VARCHAR`
	PATH.s_sid	`VARCHAR`
	PATH.s_ts	`VARCHAR`

Table 3.2: The data types we used for implementing the data models.

not require that the sensor ID is globally unique, as the object ID is. The *gid* and *gid_list* columns in tables *STAY* and *MAP* are implemented using `VARCHAR`s because, as described in Section 3.1, the *gid*s encode the hierarchy of the clusters using dots. We note that using `VARCHAR`s as primary keys is inefficient, both for disk consumption and query performance. However, we decided to keep the implementation of the data model as close to the description in [27] as possible.

We used the advisor tool of the database to determine the most appropriate indexes for all tables. We fed our workload to the advisor tool, considering the frequency with which each query should be executed (i. e., OLTP queries vs. OLAP queries) and the heavy insert workload. We used the suggested indexes in the following evaluation.

Maximal Throughput

We first examine the performance of the data staging procedure without any queries being processed in parallel. As already determined, the approaches should be able to process an arrival frequency of 500 events per second. However, as there may be peaks in the event generation, the systems must be capable of handling event frequencies greater than the expected one. Therefore, we measure the upper limit of event frequency that the database designs and the DBMS can process.

Figure 3.14 shows the maximum insert throughput for the naïve approach, the Gonzalez et al. implementation, and the RnB data model. We analyzed the performance with and without indexes, with clustered and unclustered data. Thus, we see the effect of the indexes and of the clustering on the data staging procedure. The naïve approach has a very high insert throughput (16686), since the events do not need to be transformed or pre-aggregated in any way, but are directly inserted into the database. Since this approach does not exploit the grouping of the data, clustering does not affect its performance. Indexes slow down the throughput to 9893, but it is still the highest from the three approaches. The RnB approach achieves a maximal insert performance of 2631 events per second without indexes (clustered data) and 3571 events per second when indexes are applied on clustered data. Usually, the use of indexes correlates negatively with the insert throughput, but in this case the data staging procedure benefits from the indexes as well, because of the queries it sends to the database during staging (Section 3.1.2). For the approach of Gonzalez et al., there is hardly a difference in the peak performance between the solution with indexes and without on clustered data. This is due to the fact that the main overhead during the staging procedure is not the interaction with the database, but processing the hierarchical string identifiers. As we can see, this approach is not able to achieve the throughput of 500 events per second that we expect in a medium-sized enterprise.

For the approaches RnB and Gonzalez et al., there is a difference in their behaviour when processing clustered and unclustered data. Since these two data models focus on grouping data with similar characteristics, they should benefit when the data is clustered. This was proved by our experiments: when using unclustered data, the performance of the RnB approach without indexes decreases to 170 events per second and with indexes to 1388 events per second. Analogously, for the Gonzalez et al. approach, the maximal throughput on unclustered data without indexes is 71 events per second and with indexes 70 events per second.

Query-only Workload

After determining the maximum throughput for each approach, we examine the response times of our traceability workload described in Table 3.3. We analyze the performance of each query without parallel inserts, in order to see how the inserts will affect the query performance, when conducting a mixed workload.

The workload shown in Table 3.3 consists of OLTP and OLAP queries. OLTP queries are short-running point queries that extract information about a single object. For traceability applications, it is essential to know where a certain object, e. g., a parcel, is at each point in time. This kind of queries are therefore processed very often in an RFID system. We can assume that each object is queried at least

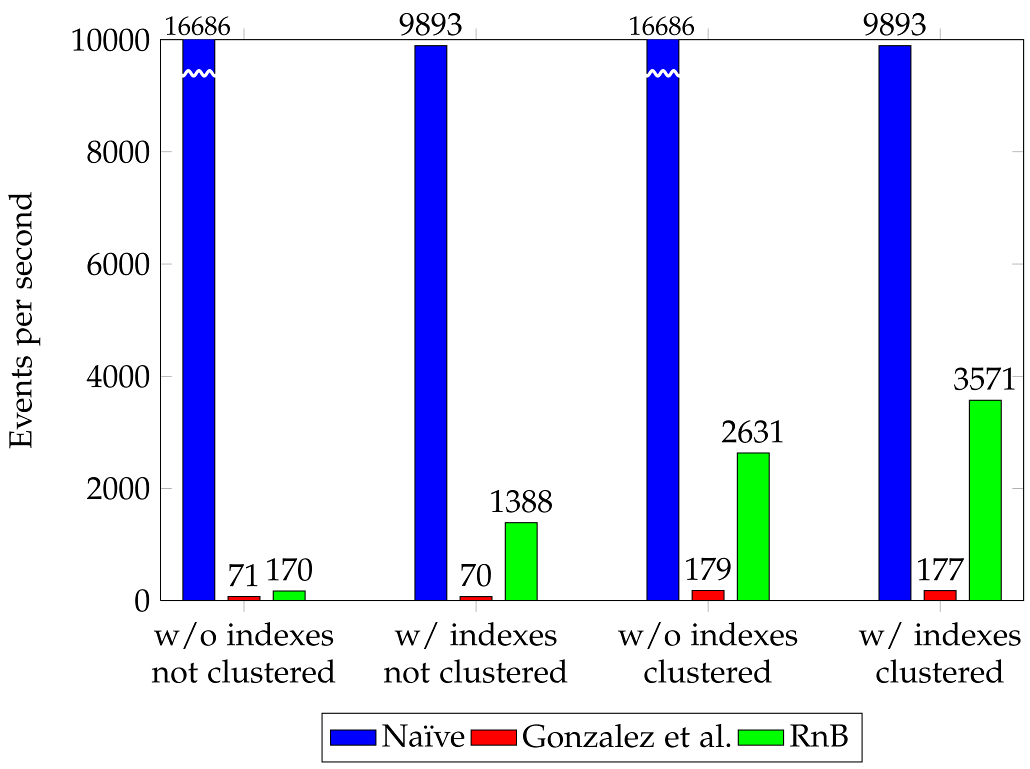


Figure 3.14: Max throughput measurement of the approaches: without indexes/with indexes, clustered/not clustered (taken from [56]). 50000 events were loaded in the data models.

once (e. g., for its last position) during its lifetime. Q1 and Q2 in Table 3.3 are OLTP queries. OLAP queries usually process a large part of the database and provide aggregated information grouped by certain attributes like sensors or divided into timeslots. They are used for report generation and thus submitted less often than OLTP queries. Q3 through Q11 are OLAP queries.

OLTP queries are executed with a 10 times higher probability than OLAP queries during the benchmark. Depending on the query type, different think times are set: OLTP clients have a think time of 1 second, OLAP clients of 30 seconds. The clients submit one query, retrieve the result, and wait for the think time before submitting the next query. We start all OLTP and OLAP clients together after the database is preloaded with 5 million events. The duration of the benchmark is 3600 seconds. Three different benchmark settings are executed: using 1, 5, and 10 query clients (OLTP and OLAP respectively). Therefore, the effect of different MPL levels on the query performance can be seen.

As already discussed for Figure 3.14, the approach of Gonzalez et al. is not capable of achieving the event throughput frequency of 500 events per second. Trying to preload the database with 500000 events using the data staging procedure took

Query	Description
Q_1	Last location of an object
Q_2	The pedigree (complete path) of an object
Q_3	The number of objects scanned by a certain sensor
Q_4	A list of objects scanned by a sensor within a time interval
Q_5	A list of objects which were scanned by sensors s1 and s2 (no order)
Q_6	A list of objects which were scanned by sensors s1 and s2 in this order
Q_7	The number of objects which were scanned by sensors s1 and s2 in this order
Q_8	A list of objects that were at sensor s, together with object x within a certain time interval
Q_9	A list of the number of objects per reader and timestamp which passed in a certain time interval
Q_{10}	A list of the number of all objects scanned by all the readers in 10 regions, ordered by region, reader, and a time interval of a second
Q_{11}	A list of the number of all objects which were scanned by the sensors s1, s2, and s3 in this order aggregated per second

Table 3.3: Queries for an RFID scenario.

more than 6 hours. Due to the inefficient staging and since this approach is not suitable for our scenario, where we assume that 500 events per second arrive at the system, we abandon this database solution from the further analysis.

Figure 3.15 shows the results of the query-only benchmark for the RnB approach. We observe that the increasing MPL level has a noticeable effect on the query performance. As expected when only one query client is used, the response time is the shortest and it increases with the number of query clients. However, for most queries the performance difference between using 5 and 10 query clients is not considerable. This means that the database system is not working to its highest capacity even when 10 query clients are running in parallel.

The OLTP queries Q1 and Q2 are short-running. They select the last position of an object and a pedigree for a particular object, respectively, and operate on the created indexes.

Q3 through Q11 are OLAP queries. Q3 calculates the number of all objects scanned by a particular reader, i. e., it counts all objects from table *READ*, which reference a path in table *PATH* containing the sensor.

Q4 determines all items scanned at a particular reader within a time interval. Similar to Q3, the query selects all objects with path IDs that contain the reader and additionally checks whether the stored timestamp lies in the correct interval. The recursive manner of this query explains the higher computational overhead compared to Q3.

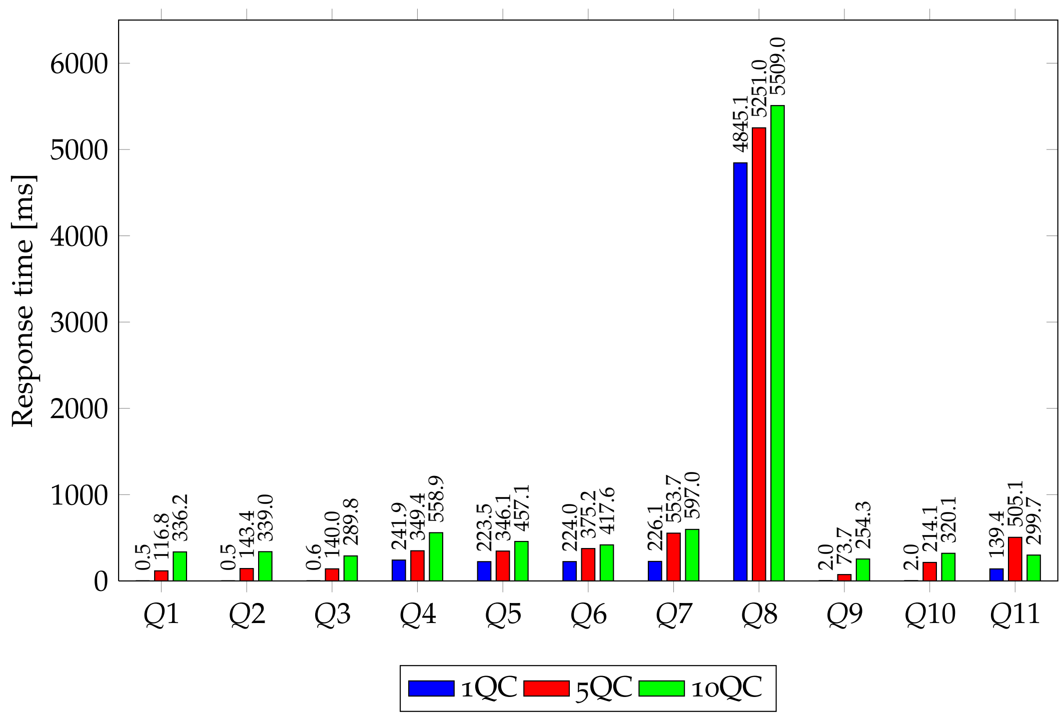


Figure 3.15: Query-only workload for the RnB approach using 1, 5, and 10 query clients (QC), respectively (taken from [56]). The database was preloaded with 5 million events.

Q5, Q6, and Q7 have a similar structure which justifies their similar execution performance. These queries find objects that were scanned at more than one location (sensor) and take into consideration the ordering of the locations. Due to the string representation of the sensor paths, we have to use string operations for calculating the right matches in the string paths. Since this locating of the sensors in the paths is applied to each of Q5, Q6, and Q7, they perform similar.

Q8 is a heavy OLAP query determining all contaminated items, i. e., all items that were scanned at the same reader as a contaminated item x and in a certain time interval. This query first returns the path of x and then checks all other existing paths if they contain any of the sensors which scanned x and whether these sensors were passed in the correct time interval. Since all entries in table *PATH* have to be examined, the performance overhead is extremely high.

Q9 lists the number of objects per reader and timestamp, which passed in a certain time interval and Q10 lists the number of all objects scanned by 10 different readers within a time interval, grouped by reader. Because these two queries do not need to determine the individual objects, but return the tuples grouped by reader or timestamp, they only operate on table *PATH* without joining it with *READ*. For this reason, they have a good performance and behave similar.

Q11 lists the number of all objects which were scanned by three different sensors in the given order within a time interval. The bigger overhead compared to Q9 and Q10 can be explained by the fact that for Q11 we need to join the *PATH* with the *READ* table in order to count the particular objects. Apart from that, the structure of Q11 is similar to that of query Q7 and as we can see they perform very similar.

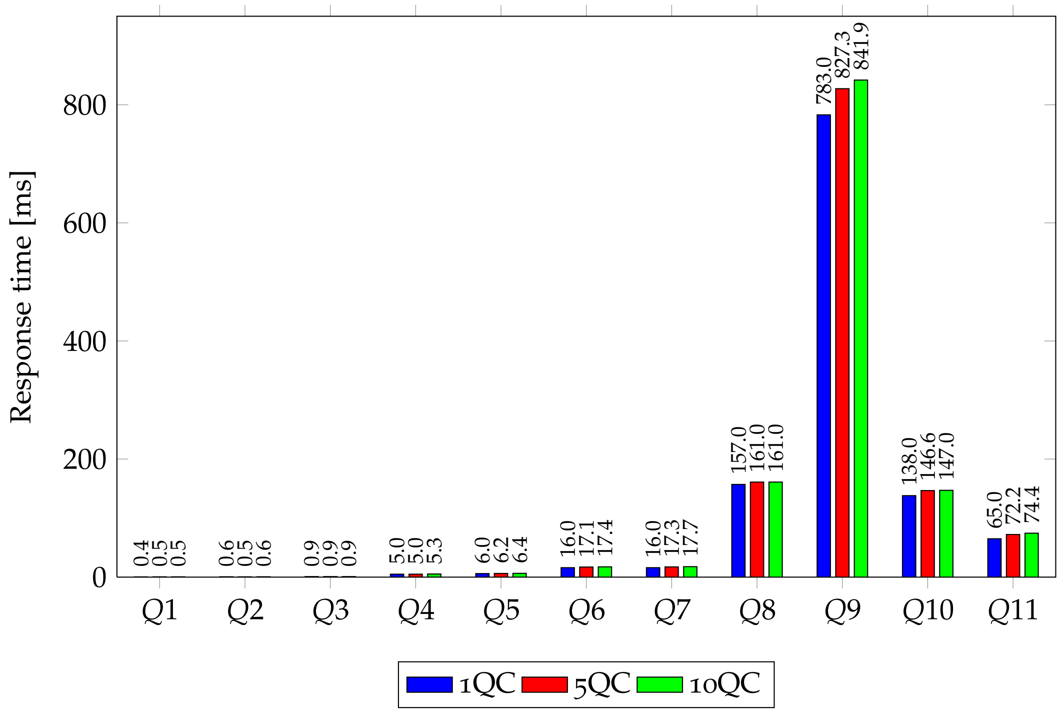


Figure 3.16: Query-only workload for the naïve approach (taken from [54]). The database was preloaded with 5 million events.

Figure 3.16 depicts the results of the query-only benchmark for the naïve approach. The benchmark settings are the same as for the RnB approach. Again, we scale the number of query clients. However, the differences between the query performance with 1, 5, and 10 clients are less than for the RnB approach. This is due to the overall shorter query response times. For the given amount of events, all queries finish in less than 1 second, which is the think time set for the OLTP queries. For this reason, the database is not overloaded.

The OLTP queries Q1 and Q2 have a similar performance to that of the RnB approach. All other queries perform better with the naïve approach, except for Q9 and Q10. None of the indexes proposed by the database advise tool was suitable for Q9, such that this query performs a table scan over the huge *EVENT* table. This explains its extremely high response time. The bad performance of Q10 can be explained by the fact that this query has to perform a self-join 10 times (for each one of the respective readers). This highlights the weak point of the naïve approach as described in Section 3.1.1: If some sort of aggregated information is needed, then this is computed at runtime and the whole table has to be processed.

Mixed Workload

A mixed workload consists of concurrent inserts and queries (OLTP and OLAP). Here, we can analyze whether the different approaches can cope with the pre-defined event generation frequency of 500 events per second when also queries are executed in parallel. The benchmark setting is analog to that of the query-only benchmark. We conduct three different runs, using 1, 5, and 10 query clients, OLTP and OLAP respectively. The database is first preloaded with 5 million events and after that the mixed workload, consisting of inserts and queries, is started. There is one insert client that produces a batch of 500 events each second and inserts them in the database using the data staging procedure.

Figure 3.17 shows the response times of the queries of the RnB approach during the mixed workload. For all of the three benchmark settings, the approach was able to manage the pre-defined frequency of 500 events per second. As expected, the query performance decreases when executing concurrent inserts compared to the query-only benchmark. Compared to Figure 3.15, the performance decreases by a factor of ca. 2. At the same time, the correlation between the query response times of the different runs (1, 5, and 10 query clients) remains the same.

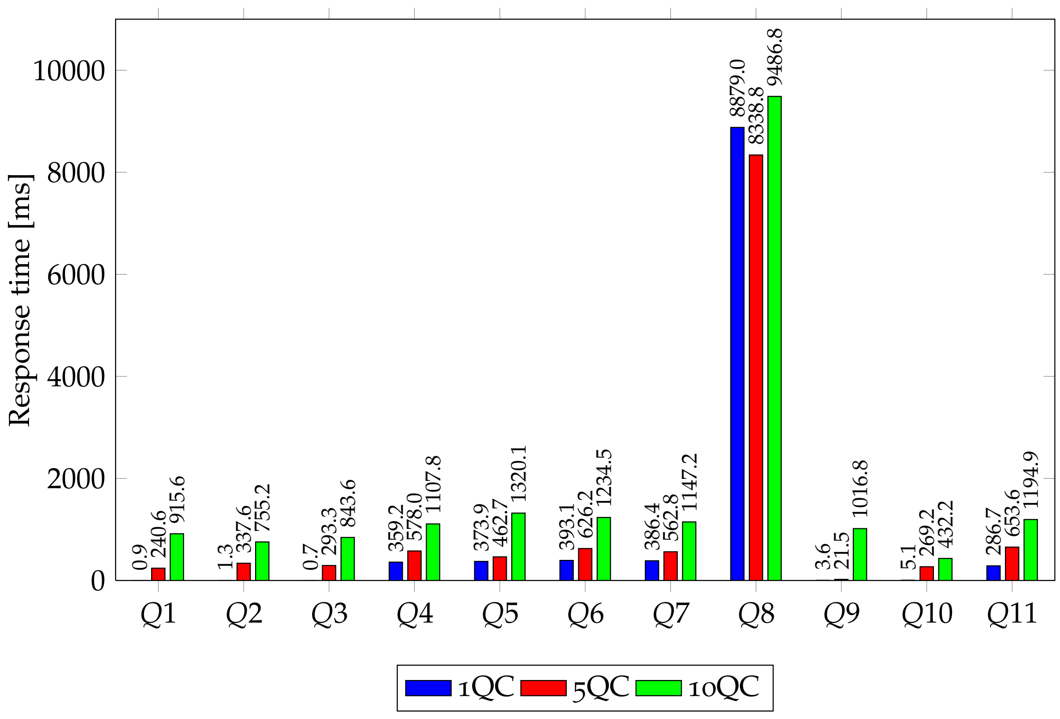


Figure 3.17: Mixed workload for the RnB approach using 1, 5, and 10 query clients (QC), respectively (taken from [56]). Concurrently, insert batches of 500 events per second are loaded into the database.

Figure 3.18 shows the query performance of the mixed workload when using the naïve approach. The approach achieves the pre-defined event generation frequency of 500 events per second. As expected, the query response time is affected by the concurrent inserts and is approximately a factor of 2 worse compared to the query-only benchmarks. However, except for Q9, the overall query performance of the naïve approach outdoes the RnB approach. The reason for the overhead of Q9 is the same as for the query-only workload: none of the indexes proposed by the database advise tool was suitable for Q9. The good overall performance of the naïve approach is due to the very efficient staging procedure, which barely affects the query performance. The staging procedure of the RnB approach is much more time-consuming and thus has a negative influence on the query performance.

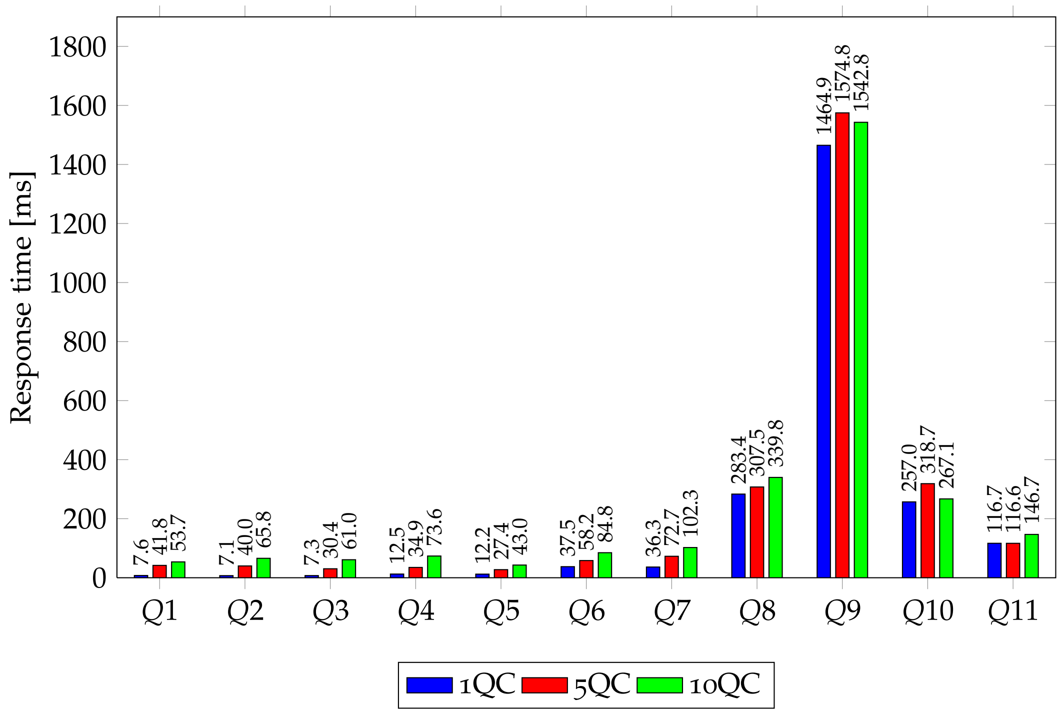


Figure 3.18: Mixed workload for the naïve approach using 1, 5, and 10 query clients (QC), respectively (taken from [54]). Concurrently, insert batches of 500 events per second are loaded into the database.

3.3.4 Evaluating the Bloom Filter Approach

We present experiments for comparing the naïve approach, that had the best overall performance from the existing approaches, and the Bloom filter approach implementation on a commercial database. The results show that our approach succeeds in handling a continuous event stream as expected in a medium-size business and even outperforms the naïve approach in query processing.

Experiments

We report benchmark results for experiments conducted on a commercial database implementing the Bloom filter approach and the naïve approach. The database runs on a 64 bit-Red Hat Enterprise Linux server with two Intel Xeon 3.16 GHz CPUs, 8 GB main memory, and 8 SAS disks associated with RAID level 5. This is a different experimental environment from the test environment for the existing RFID approaches. Thus, the experiment results are not directly comparable.

Data Staging

We first examined only the data staging procedure of our approach without any queries being processed in parallel. As we motivated above, a suitable system has to be able to handle an average data arrival frequency of 500 events per second. We thus ran benchmarks with this fixed event generation frequency and found out that the different data models and database systems are able to cope with the arriving events in the data staging process. However, as there might be peaks in event generation, we analyze the upper insert limit of the database designs. The naïve approach has a very high insert throughput (15466 events per second), since the events do not need to be transformed in any way, but are directly inserted into the database. The Bloom filter implementation has a throughput of only 2240 events per second due to the overhead of the Bloom filter processing during data staging. The conclusion would be to fall back on the naïve approach in periods of very high loads. However, the query response times will show that the naïve approach does not support efficient query processing when the database grows in size and therefore it is not an appropriate long-term solution.

Mixed Workload

We also analyzed whether the specified frequency can be kept while executing a mixed workload consisting of concurrent inserts and OLTP and OLAP queries. Here, the workload is designed as follows: two insert clients continuously insert events during one hour, thus generating a total of 1.8 million events. The query clients start submitting queries after the benchmark has been running for 5 minutes, so that approximately 150000 events are preloaded before the first query arrives at the database. Each query type (OLTP and OLAP) is handled by one query client. Depending on the query type, a think time of 1, respectively 60 seconds is set up for OLTP and OLAP queries. The clients submit one query, retrieve the result, and wait for the think time before submitting the next query.

Figure 3.19 shows the average response times of the OLTP queries. The Bloom filter approach has a better response time for Q1. The reason for this is that in general the *OLTP* table contains considerably less data than the *EVENT* table of the naïve approach, because it stores only the most current event per object. Still, determining the last position of an object (Q1) has nearly the same response time for both approaches, since the difference in the size of both tables is not considerable for the given amount of data.

For Q2, the Bloom filter approach is a factor 2 slower than the naïve approach. This is due to the Bloom filter processing, which requires a two-step communication of the application and the database for determining the relevant regions and querying the corresponding tables.

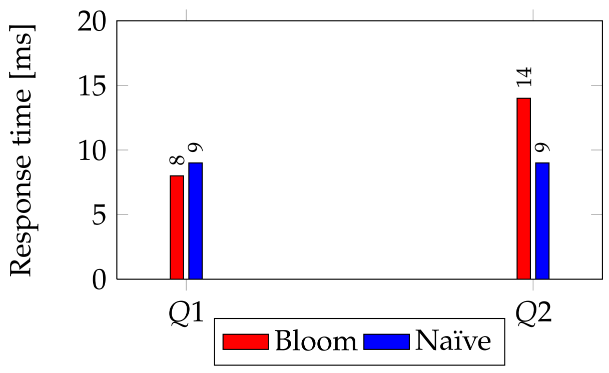


Figure 3.19: Mixed Workload: OLTP Queries

The response times of the OLAP queries are presented in Figure 3.20. The queries Q3 and Q4, which operate on one particular reader, are much more efficient for the Bloom filter approach, because its data is segmented by sensor, which results in smaller tables per reader, compared to the centralized big table of the naïve approach. This means that once we know in which region the particular reader is located, we can operate only on the data of this table. In our case, each one of the *REGION* tables contains one tenth of the data of the naïve approach.

Q5, Q6 and Q7 operate on two distinguished readers. These can reside in one region, or in two different regions. In both cases, the approach processes less data than the naïve approach and this affects its execution time.

Q8 is a heavy OLAP query – the contamination query. It determines the objects that traveled together with a contaminated object in a certain time interval. For the Bloom filter approach, we first look up the Bloom index for the contaminated object and query then the possible contaminated regions for all objects in the contaminated time interval. Again, if the contaminated object was located in only one region, the query is extremely efficient.

Q9 groups information per reader and timestamp for a particular region and outperforms consequently the naïve approach.

All shown queries except query Q10 are executed a factor of 20 slower on the naïve approach schema. This is due to the much higher amount of data the naïve approach has to process for each query. Query Q10 takes half the time on the Bloom filter database schema. The query processes a union over 10 subquery results.

Q11 selects all objects that were scanned at three distinguished readers. Again, the Bloom filter approach takes advantage if the data is located in one *REGION* table.

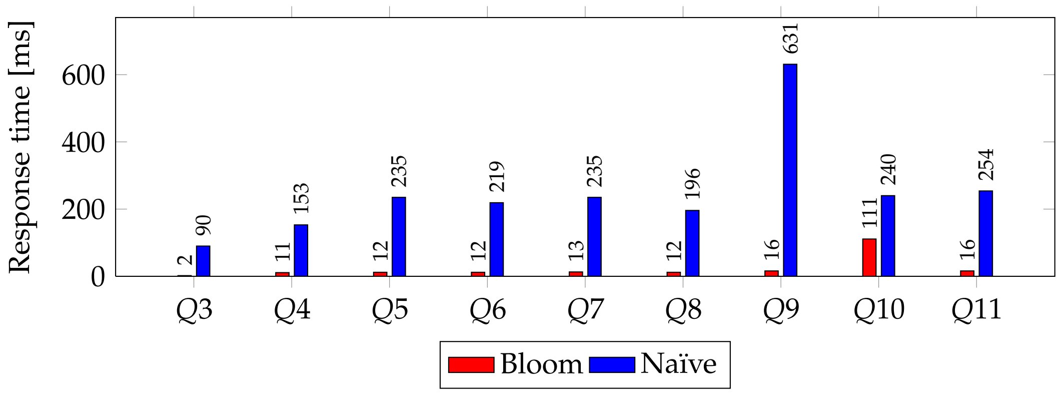


Figure 3.20: Mixed Workload: OLAP Queries

Overall, we can conclude that by the smart partitioning of the big *EVENT* table by reader, which results in the Bloom filter database design, we achieve an approach that can cope with the insert frequency of 500 events per second and additionally has a considerably better query performance. Because of the partitioning, this approach processes often a small portion of the data, instead of working on the complete event data. A disadvantage of the Bloom filter approach is however that for some queries like, e. g., Q2, a two-step query processing is needed. First the value of the Bloom filter is determined and then the corresponding *REGION* table is queried. This drawback can be overcome if the Bloom filter index is integrated in the database system.

3.4 Related Work

Many related RFID approaches [27, 35, 37] are primarily focusing on appropriate data schemas for RFID data in relational databases. We addressed these approaches in detail in Section 3.1. We compared them qualitatively by classifying them according to the criteria defined in Section 3.3 and quantitatively by the conducted benchmarks.

In RFID data management, the cleaning and filtering of the raw data is an important pre-processing step. Since readings are still considered generally inaccurate and error-prone, in the context of traceability data, handling tags being missed (so-called false-negative readings) or unexpected extra readings (false-positive readings or "noise") is challenging [52]. In [31], a declarative and adaptive smoothing filter for RFID data (called SMURF) is proposed. SMURF controls the window size of the smoothing filter adaptively using statistical sampling. A recently published inference approach for accurate estimates of object locations under consideration of the objects' containment relationships is presented in [12]. This approach outperforms SMURF's accuracy as discussed in [12]. Data cleaning and outlier detection are also addressed in [6, 13, 38]. Data cleaning is not in the scope of this work. We assume data cleaning was performed as a pre-processing step using one of the existing approaches.

3.5 Summary and Conclusions

In this chapter, we first presented existing database solutions for efficient management of RFID data. These approaches were implemented and further analyzed. Second, we propose a new approach, which is designed such that the OLTP and the OLAP part reside in one database in contrast to the other approaches. Finally, we evaluate the approaches using three different benchmarks: maximal throughput, query-only evaluation, and a mixed workload consisting of inserts and queries. We examine whether the approaches are able to manage the predefined event frequency of 500 events per second. Among the existing approaches, the naïve has the best overall performance. We compare it further to our Bloom filter approach and find out that the latter outperforms the naïve approach with respect to most of the queries' response times.

The Bloom filter approach accelerates the query processing for the typical traceability scenario queries. It has to be considered that there is a trade-off between the event processing throughput and the query processing, so that the Bloom filter approach gives something from its performance up in terms of the maximum achievable event processing throughput. It pays off, however, when it comes to query processing. The performance advantages in terms of shorter query response times can be attributed to the following factor: the splitting of information is beneficial for queries occurring in a typical traceability scenario. As most of the queries are interested only in a particular segment of the data, e.g., the behavior of objects grouped by sensors, they take advantage of our hierarchical structure. Determining the pedigree of an object using the Bloom filter results in two phase transformation. If this could be proceeded in one step, we would be able to make a better use of the Bloom filter and achieve a better performance.

4 A Dedicated Triple Store for RFID Data Management

4.1 Motivation

Until now, we only considered traditional DBMS approaches for the management of RFID data. In this chapter, we advocate our decision for going for a dedicated solution for efficient RFID data management.

4.1.1 From a Traditional DBMS Approach to a Dedicated Solution

As described in Chapter 3, there are several related research efforts focusing on providing solutions for efficient management of RFID data residing in relational databases. The data model of Gonzalez et al. [27] is a typical warehouse approach, but it doesn't support an efficient data staging process. In our experimental benchmarks, we found out that the throughput was only ca. 170 events per second without queries running in parallel. The approach of Krompass et al. [35] is a hybrid approach consisting of a main memory resident table for recent events and a warehouse data store for historical information. In our benchmarks, the event processing throughput of the system was ca. 500 events per second, with concurrent queries. The "prime number" approach of Lee and Chung [37] can only be applied if the object movement is known in advance and thus it is not possible to incrementally update this database solution. Following the state-of-the-art in managing RFID data, in our first approach [20] we manage traceability data using a traditional relational DBMS. We developed a schema that considers the characteristics of traceability data and stores the OLTP and OLAP data in the same system. We used a Bloom filter as an index in order to efficiently reconstruct the path of an object. We explained and compared the approaches in detail in Chapter 3. Even with the Bloom filter approach we could not fully exploit the advantages of our schema as the Bloom filter is not implemented as part of the relational DBMS. Therefore, some queries need a two-step communication between the application and the database system and this affects their run time negatively. After analyzing the limits of the presented traditional DBMS approaches for efficient management of RFID data, we choose a different way of proceeding. We followed the argument of Stonebraker that the "one size fits all" era in database design comes to an end [49], and show that dedicated (customized) systems outperform the generic mainstream systems.

In this chapter, we present a scalable dedicated solution for efficient storage and management of RFID data, **the RFID Triple Store**, which is inspired by recent work on Resource Description Framework (RDF) triple stores [40]. The RFID Triple Store supports an efficient data staging and fast OLTP and OLAP query processing for traceability data. It further provides a sustained throughput of 2500 events per second, which satisfies the requirements stated in Chapter 2 for a world-wide enterprise.

We begin with a short introduction of RDF data and explain the similarities and differences between RDF and RFID data. Based on these findings we design the RFID Triple Store as a dedicated storage solution for RFID data.

4.1.2 A Short Introduction to RDF

The Resource Description Framework (RDF) [3, 34] is a family of W3C standards which provide a model for data interchange on the Web. It represents schema-relaxable or even schema-free structured information in XML syntax. RDF is being used in the context of Semantic Web data for ontologies and knowledge bases, in life sciences for e-science data repositories, and also for Web 2.0 platforms and information mash-up applications.

RDF makes statements about Web resources in form of expressions, called RDF triples

$$(\textbf{\textit{subject, predicate, object}}).$$

The *subject* denotes the resource, and the *predicate* expresses the subject's attributes and/or the relationship between the subject and the *object*. Subjects and predicates are Uniform Resource Identifiers (URIs). Objects, however, can be URIs or literals. The set of predicate names can be quite diverse and although they often resemble attributes there is no global database schema.

For example, information about this thesis can be represented by (at least) these triples

$(\texttt{id}_\texttt{p},\texttt{hasTitle,"Efficient ..."})$
$(\texttt{id}_\texttt{p},\texttt{hasAuthor},\texttt{id}_\texttt{a})$
$(\texttt{id}_\texttt{a},\texttt{hasName,"Veneta Dobreva"})$.

Here, the subjects $\texttt{id}_\texttt{p}$ and $\texttt{id}_\texttt{a}$ as well as the predicates `hasTitle`, `hasAuthor`,

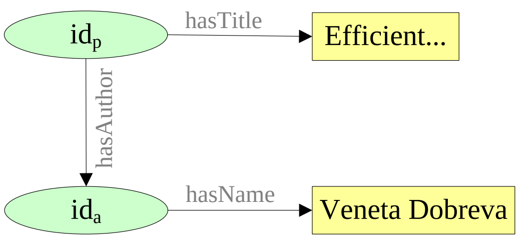


Figure 4.1: Simple RDF example.

and `hasName`, and the object `id`$_a$ are URIs whereas the objects `"Efficient ..."` and `"Veneta Dobreva"` are literals.

The relationships between subjects and objects that are stated by the predicates, result in a highly interconnected graph. The RDF graph data model is composed by vertexes – the different subjects and objects (called entities), and edges, representing the relationships between them. Therefore, the information about an entity is depicted by a directed named edge ending in another entity vertex or in a special literal vertex, containing the concrete value. The RDF graph corresponding to the example above is shown in Figure 4.1. A real example of an RDF graph from the DBpedia RDF dataset [2] is shown in Figure 4.2. The example has been simplified by omitting the long URIs in order to make it more readable. One sample relationship is the entity "Woody Allen", which has an occupation called "Director" and is born in "Brooklyn". As we see, RDF graphs can be highly branched and interconnected.

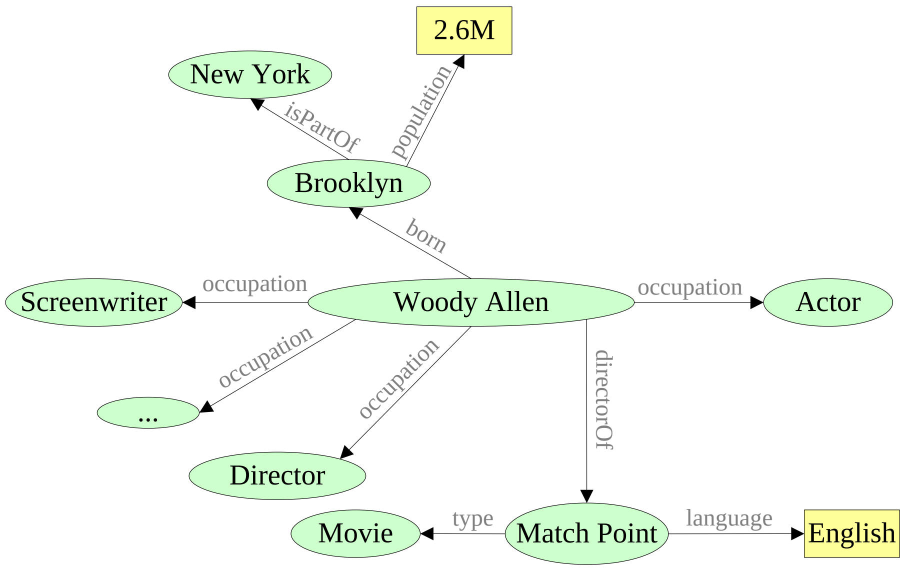


Figure 4.2: Example RDF graph data from DBpedia [2]. Predicates are represented by oval shapes and subjects by rectangle shapes.

RDF data is retrieved using the SPARQL [4, 44] query language. SPARQL resembles query by example in that each predicate which the result data has to fulfill is expressed as an RDF triple in which queried data and equi-joins are represented by variables.

The following example:

```
select ?p where {
  ?x <hasTitle> ?p.
  ?x <hasAuthor> ?a.
  ?a <hasName> "Veneta Dobreva"
}
```

requests the titles of all documents written by Veneta Dobreva. As we can see, `?p` is bound to titles, and `?x` and `?a` are equi-join predicates. In this example, all predicates (i. e., `hasTitle`, `hasAuthor`, and `hasName`) as well as one object `"Veneta Dobreva"` are given. In order to evaluate the query, at least the three triples given above have to be considered: `?a` is bound to $\texttt{id}_a$, `?x` is bound to $\texttt{id}_p$, and `?p` is bound to `"Efficient ..."`. Further documents written by Veneta Dobreva are copies of the first triple only differing in the object (title) so that `?p` is bound to different objects.

Due to the triple structure, variables can occur in multiple patterns. Queries will inherently contain long chains of selfjoins and large join stars over many-to-many relationships. At the same time, query optimization is not possible as join attributes are difficult to predict. Further, join order optimizations require data statistics for selectivity estimation, but in the absence of a schema a suitable granularity of statistics gathering is non-trivial. The evolving structure of the data and the variance and dynamics of the workload make it unattractive to rely on an auto-tuning approach. These properties of RDF pose technical challenges for efficiently managing and querying RDF databases. There has been a significant research effort in this field as discussed in Section 4.6. We were inspired by one of these works – the RDF system RDF-3X, which stores and queries RDF data extremely efficiently, and is found to be the fastest RDF store [30].

4.1.3 Similarities and Differences between RFID and RDF Data

The obvious similarity between RFID and RDF data is first of all the triple structure of the data: an RDF triple is represented in the form $(subject, predicate, object)$ and RFID data in the form (e, r, t) as described in Section 2.1.3. Second, in both scenarios there is a high volume of data that has to be dealt with. Typical RDF databases are, for instance, huge reference repositories in life sciences. The high event generation frequency of RFID data leads to fast-growing RFID archives of multiple terabytes. Further, passed or historical RFID events resemble RDF read-only databases, where no changes are expected. Since the timestamps of the RFID events are growing monotonically during event generation, old events will never

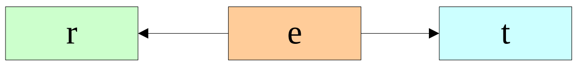


Figure 4.3: RFID event as a graph.

need to be updated, i. e., they are static. Therefore, an RFID database will be of an append-only manner.

However, there exist several important differences between RFID and RDF data. RDF uses a graph data model that represents entities and their relationships as shown in Figure 4.2. An RDF dataset usually results in a big highly interconnected graph, whereas, if we represent an RFID dataset as a graph, it will consist of multiple small "star" pattern graphs for each single event. Given an RFID event triple: *(e, r, t)*, *e* will be the central entity connected to *r* and *t* respectively, as shown in Figure 4.3. Further, due to the fact that the three components of the RFID triples are semantically independent, their value domains are disjunct and they are never combined during query processing in a join predicate or in a comparison (e. g., we will never join a reader ID with a timestamp). Reasonable joins in an RFID scenario combine a reader ID with another reader ID, since we are often interested in information about objects that travel a certain path. Because the time factor is of a particular importance for traceability data, timestamp joins are extremely important. However, not equi-joins are applied in this context, but joins over a time window, the so called range queries, which determine information within a certain time scope. These features are leveraged in query processing for the RFID Triple Store. Another important difference is that the RFID traceability data is very dynamic and requires efficient data staging mechanisms, whereas RDF data is updated less frequently and the RDF stores are primarily focused on query processing performance.

We address the challenges posed by RFID data described in Chapter 2 and the characteristics discussed in this chapter in the design of our dedicated system as follows: (1) we incorporate elaborated indexing techniques leveraging the specifics of RFID data, in order to enable efficient event processing; (2) the query engine takes advantage of the RFID characteristics (e. g., the monotonic increase of timestamps) to speed up query processing. Our implementation of the RFID Triple Store builds on the code base of the open source RDF-3X engine [40], which has been claimed to be the fastest RDF store [30].

4.1.4 Contributions

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

- We introduce the architecture of a dedicated system for efficient RFID storage and management.
- We design RFID-specific indexes for efficient event processing.

- We provide three different data dictionaries for encoding the long identifiers of the three components of the triple format *(e, r, t)*. The monotonic increase of timestamps implies an order-preserving timestamp dictionary.
- We use the specific RFID properties (e. g., timestamps increase monotonically) to optimize the query engine of the Triple Store.
- We experimentally evaluate our system using a mixed workload consisting of inserts (data staging process) and queries (OLTP and OLAP) and analyze the sustained throughput that can be maintained. Further, we compare our approach against a commercial row-store and a non-commercial column-store database system.

4.2 Triple Store Architecture

The RFID Triple Store is an "RFID-aware" database, which achieves high performance by leveraging characteristics specific to traceability data in the design and implementation of its index structures and query engine. In the following, these characteristics are described and the key design principles behind the system are presented. Some aspects in this chapter are examined in the work of Robert Brunel [11] which was supervised by the author of the thesis. Parts of this work were also published in [21].

4.2.1 Triple Store Indexes

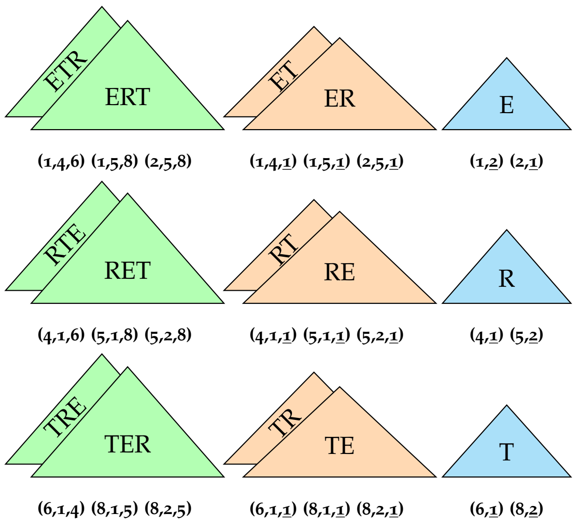
The RFID Triple Store implements a logical schema for RFID data, which stores all event triples in a single large table with columns E, R, and T for EPC, reader, and timestamp. This direct representation obviates the need for non-trivial preprocessing steps that many related approaches suffer from (see Section 3.1). Further, we create an extensive set of indexes, shown in Figure 4.4, that are beneficial for query processing:

- Full Triple Indexes for any permutation of the three columns (ERT, ETR, RET, RTE, TER, and TRE)
- Aggregated Triple Indexes for any permutation of any column pair (ER, ET, RE, RT, TE, and TR)
- Fully-Aggregated Triple Indexes for any single column (E, R, and T)

The Full Triple Indexes store full triples redundantly, rather than pointers to the triples table. These indexes are essentially reordered copies of the triples table, which becomes dispensable. Consequently, the Triple Store processes all queries using the indexes only, and does not materialize the triples table explicitly. Having all possible orderings of the triples is beneficial during query processing (as explained in Section 4.4).

EVENTS		
E	*R*	*T*
1	4	6
1	5	8
2	5	8

(a)



(b)

Figure 4.4: (a) RFID events (b) RFID Triple Store indexes. For ease of presentation the value ordering of only one permutation of the index type (Full, Aggregated, and Fully-Aggregated) is depicted. The underlined values denote a counter for the number of aggregated triples (explained in 4.2.1).

The indexes are implemented as clustered B$^+$-trees. The triples in the indexes are sorted lexicographically by (v_1, v_2, v_3), where v_i denotes the value of the *i*-th column. The three types of indexes store (v_1, v_2, v_3) tuples, $(v_1, v_2, count)$ tuples, and $(v_1, count)$ tuples, respectively, where *count* denotes the number of aggregated triples with equal key values. In Figure 4.4, the count values for the Aggregated Indexes are shown underlined, e. g., $(1,\underline{2})$ in the Fully-Aggregated Index E depicts that the object with EPC 1 was scanned twice. To see how these counts can be useful in query processing, consider the query "How many objects passed reader *r* yesterday?": One possible execution plan would select all triples related to *r* from the RT index and then sum up all *count* values for yesterday's timestamps. Aggregated and Fully-Aggregated indexes are much smaller than the Full indexes so that their size can be neglected for the total database size. In general, the size of all indexes together is less than the size of the original RFID data. This is achieved through the index compression, which is described in the next section.

The B$^+$-tree Data Structure

All indexes in the RFID Triple Store are represented by clustered B$^+$-trees, which have the additional property of their leaf node entries to be compressed. The compression is not applied on each single leaf node entry, but on all entries of one page. In particular, compression is not deployed across page boundaries. This is explained in detail in Section 4.2.2. As known, the B$^+$-tree data structure supports lookups of data values for a particular key value. Its main characteristic is that all its nodes are database pages. An entry on an inner node holds a key value and a pointer to a child page. The leaf nodes are forward-linked pages, where all values are sorted in key order. This enables very efficient range scans for our B$^+$-tree with optional start and stop conditions (they specify the lower and upper bound of the scanning scope). These start and stop conditions are highly exploited during query processing. They speed up the range queries over a time window that are typical in an RFID scenario.

4.2.2 Index Compression

The level of redundancy generated by the 15 indexes is affordable due to three index compression techniques used in the RFID Triple Store: data dictionary compression, prefix compression, and difference compression. In the following, each of them is described in detail.

Data Dictionary

By employing a data dictionary, repeated information (typically long strings) in the data is replaced by a short, unique code. This reduces the used storage and speeds up processing. RFID data is very suitable and benefits strongly from dictionary encoding due to the following facts:

1. **Values are long character sequences**
   EPCs are long identifiers encoding product groups, producers, a single product's serial number and other information in up to 198 bits (as defined by the SGTIN-198 standard [23]). For our scenario, we apply the commonly used SGTIN-96 EPC standard. Readers may also include their ID and additional information (like GPS position) in the generated event. Timestamps contain at least date and time, however additional information such as a time zone might be provided.

2. **Values occur multiple times**
   Depending on the application scenario, every EPC occurs up to hundreds of times. The reader infrastructure is usually stable so that the same reader will appear very frequently. Each reader will produce on average $1/x$ of the events, where x is the number of readers, if a uniform distribution is assumed. Depending on the event generation frequency and time units used within the infrastructure, each timestamp recurs in a high number of events.

Consequently, by mapping EPC, reader ID, and timestamp values in each triple (v_1, v_2, v_3) to internal numerical IDs (id_1, id_2, id_3) using a data dictionary, we can

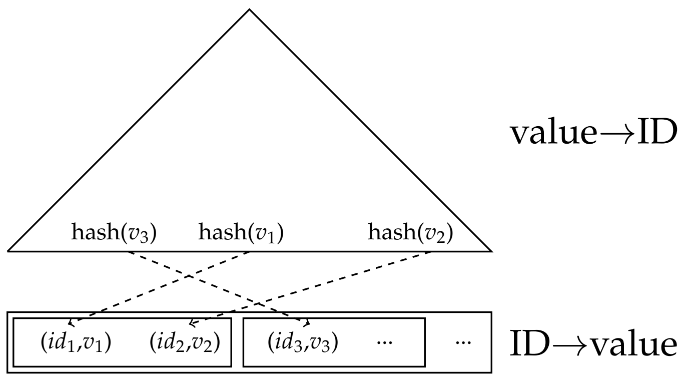


Figure 4.5: Data dictionary in the RFID Triple Store.

substantially reduce the storage space in the RFID Triple Store and speed up the processing.

We provide one data dictionary for each of the triple values. This is based on the fact that the three columns of *E*, *R*, and *T* are semantically independent, i. e., they are never combined during query processing in a join predicate or a comparison (e. g., we will never join an *E* with an *R* column or compare an *E* with an *R* column). Each dictionary can thus be optimized for "its" value type (*E*, *R* or *T*). Each dictionary assigns numbers starting from 0, which results in contiguous sequences of IDs without gaps. Index compression, in particular prefix compression (that is additionally applied and explained below), benefits from both, lower absolute and relative ID values. It is therefore desirable in an RFID scenario to maintain the three ID domains as disjunct ranges, so that the IDs are as small and as similar (the numbers are not wide apart) as possible. For T indexes, particularly, IDs grow monotonically with time, and sorting by time is essentially sorting by ID, which we leverage in query processing.

For the efficient mapping from value to ID and vice versa, we maintain two index structures, which are illustrated in Figure 4.5. Assuming that all values of a certain type (EPC, reader, and timestamp) have a common length, the entries in each dictionary have a fixed size. This allows for efficiently mapping from ID to value in one step by the use of a direct mapping index [22]. Direct mapping is a technique used to efficiently map logical OIDs (in our case dictionary IDs) to a physical address (in our case the address of the real triple value). It is more robust than mapping logical identifiers via hashing or B$^+$-trees and outperforms both methods, as stated in [22]. The mapping index is implemented as a sequence of ID-value pairs stored on subsequent pages. Accessing the *i*-th element ($0 <= i <$ number of entries in the dictionary) can be done quickly using an in-memory list of page blocks ("chunks") and some arithmetics.

To efficiently support the reverse mapping from value to ID, each dictionary maintains a B$^+$-tree that maps the hash of a value to a set of candidate page numbers within the mapping index. These pages are then searched for the value with the respective hash. We use the hash values of the strings instead of the long character sequences, in order to speed up the search operations in the B^+-tree and to reduce the space consumption of the dictionary on disk.

One can use a hash index instead of B^+-tree for managing of the reverse mapping. There is however a trade-off between the access speed and the insert rate into the data structure. Since RFID data is not static, but highly dynamic, there will be continuous inserts. Hash tables do not perform well when there is a big amount of inserts so that they have to be often extended. For this reason, the B^+-tree is the better choice in this case.

Prefix and Difference Compression

Tuples that share a common prefix benefit from a technique called prefix compression. In the indexes RET, RTE, RE, RT, and R, most triples share the same first triple value, the reader ID. For EPCs and timestamps the same is true, albeit less pronounced. The ID tuples of the RFID Triple Store are stored in a lexicographical order and therefore neighboring elements are usually very similar (e. g., EPCs are assigned in ascending order and time values are monotonically increasing). Thus, for the Full Triple R Indexes, most neighboring ID triples have equal id_1 values. The Full Triple E and T Indexes have only slightly different id_1, id_2, and id_3 values. Depending on the data generation (the path length of an EPC) and on the event generation frequency, the E and T indexes benefit from an equal id_1 value as well. This observation leads to the idea of the difference compression: storing only *changes* between IDs rather than "full" ID values. The tuples $(1, 4, 7)$, $(1, 5, 8)$, and $(1, 5, 9)$ will be therefore compressed to $(1, 4, 7)$ $(-, 1, 8)$ $(-, -, 1)$, where "$-$" denotes prefix compression and the value 1 represents the difference to the predecessor value. If a value changes according to its predecessor, then the following triple values are not difference-compressed.

The ID values as well as the *count* values of the data tuples to be compressed are 4 byte unsigned integers. A full triple as well as an aggregated triple has three, and a fully-aggregated triple has two such values. The differences ("deltas") between a tuple component and its predecessor consume between 0 and 4 bytes per value, as only the non-zero tail bytes are written and leading zero bytes are skipped. Per tuple of deltas, one header byte with size information is required to encode (and later reconstruct) the total number of bytes used by the deltas. Remaining unused bits in the header byte are used for an extra compact encoding, in case only v_3 changes for a triple and the delta is less than 128 (as is very common). The compression used for Aggregated and Fully-Aggregated Triple Indexes has minor differences: Most changes involve a gap in v_2 for aggregated triples or in v_1 for fully aggregated triples, together with a low *count* value. More details on the algorithms are given in [40].

Prefix and difference compression are applied to all of the index B^+-tree leaf pages, but not to inner nodes. Using compression in inner nodes would make it impossible to use binary search for keys. Also, in order to preserve the properties of the standard B^+-tree, the index pages are always compressed and decompressed as a whole, and compression is not applied across page boundaries. Therefore, the first triple on a page is stored uncompressed and is used as an anchor.

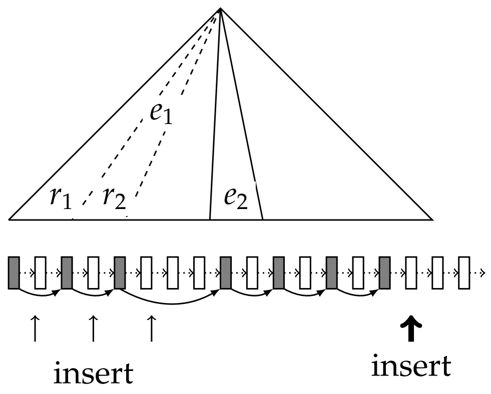
Index	**ERT**	**ETR**	**ER**	**ET**
Items:	10^7	10^7	9901313	10^7
Pages:	3016 (12225)	2379 (12225)	1213 (7260)	1425 (7332)
Items per Page:	3315 (818)	4203 (818)	8162 (1364)	7017 (1364)
Item Size:	4.0–5.4 B	3.8–4.8 B	1.9–2.3 B	1.89–2.5 B
Compression Time:	197 ms [65 μs]	241 ms [101 μs]	205 ms [169 μs]	186 ms [130 μs]
Decompression Time:	325 ms [107 μs]	105 ms [44 μs]	84.3 ms [69 μs]	81.4 ms [57 μs]
Index	**TRE**	**TER**	**TR**	**TE**
Items:	10^7	10^7	9759253	10^7
Pages:	2968 (12225)	2325 (12225)	840 (7155)	839 (7332)
Items per Page:	3369 (818)	4301 (818)	11618 (1364)	11918 (1364)
Item Size:	4.0–5.0 B	3.8 B	1.4–1.5 B	1.3–1.4 B
Compression Time:	208 ms [69 μs]	231 ms [99 μs]	137 ms [162 μs]	102 ms [121 μs]
Decompression Time:	95.1 ms [32 μs]	104 ms [44 μs]	62 ms [73 μs]	62.5 ms [74 μs]
Index	**RET**	**RTE**	**RE**	**RT**
Items:	10^7	10^7	9901313	9759253
Pages:	2999 (12225)	2972 (12225)	1743 (7260)	1738 (7155)
Items per Page:	3334 (818)	3364 (818)	5680 (1364)	5615 (1364)
Item Size:	4.7–5.0 B	4.7–5.0 B	2.8–3.0 B	2.8–3.0 B
Compression Time:	191 ms [63 μs]	202 ms [67 μs]	198 ms [113 μs]	207 ms [119 μs]
Decompression Time:	89.4 ms [29 μs]	164 ms [55 μs]	72.7 ms [41 μs]	115 ms [66 μs]
Index	**E**	**R**	**T**	
Items:	501783	1000	200094	
Pages:	73 (246)	1 (1)	36 (98)	
Items per Page:	6873 (2046)	1000 (2046)	5558 (2046)	
Item Size:	2.3–2.4 B	3.0 B	2.9–3.0 B	
Compression Time:	10.6 ms [144 μs]	26 μs	3.84 ms [106 μs]	
Decompression Time:	3.49 ms [47 μs]	13 μs	1.62 ms [44 μs]	

Table 4.1: Results for the prefix compression benchmark test (adopted from [11]). For time measurements, the values in square brackets are the average per-page values derived from the accumulated times. For page and item counts, corresponding numbers of uncompressed data are given in round brackets for comparison. The two given item sizes per index are the minimum and maximum of the set of average item sizes on all its compressed pages.

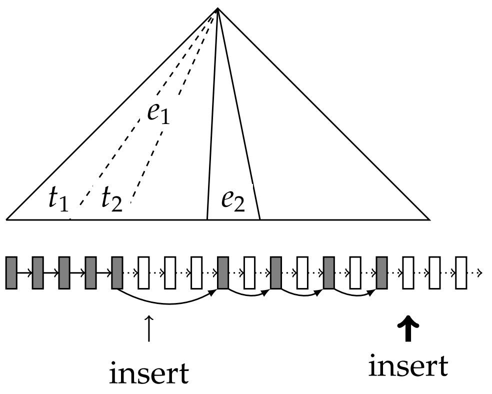
We found out that data dictionary, prefix and difference compression reduces the size of the indexes to the factor of four (see Table 4.1) compared to using uncompressed indexes. The Aggregated and Fully-Aggregated indexes are much smaller than the Full Triple Indexes and the increase of the total database size due to these indexes is negligible. These observations can be proved in the following experiment. We measured the compressed data sizes for every index as well as the accumulated time needed for all compression operations and all decompression operations. The results are summarized in Table 4.1. The test illustrates a typical mini RFID scenario: it is conducted with a set of 10^7 ID triples, i. e., events, equally distributed over the 1000 RFID sensors and read from a pre-processed file with size of about 78.3 MB. From this set of triples, 15 sequences corresponding to the 15 different indexes (with different permutation order and aggregation levels) are built. For each index, all triples are subsequently compressed and packed into chunks of 16 KB (matching the size of a page) and then decompressed again. For every index, the total number of items, the pages used (the occupied pages without compression are given in brackets), items per page (in brackets without compression), the item size (minimum and maximum value), the compression and decompression time (in brackets the calculated time per page) are given. We observe that the compression factors depicted in the table can be considered good, but significantly vary from index to index. They depend not only on the type of index (Full, Aggregated or Fully-Aggregated) but also on the ordering of the data within the index (RET, RTE, ...).

The compression factor is in the range of 19-24% for the Full, 11-24% for the Aggregated, and 30-37% for the Fully-Aggregated Triple Indexes. The R index consists of only one page, since we presume 1000 different readers in our scenario, which together take only 3000 bytes. This is realistic since in a real-world RFID scenario the number of sensors that are applied in the infrastructure is usually not more than several hundreds. For some indexes (ERT and ETR), the minimum and maximum average item size differ by more than 30%. This is caused by the fact that the compressed sizes of absolute EPC and timestamp IDs increase slightly, that means from 2 to 3 bytes once they exceed 2^{16}, i. e., when there are more than 65535 items from each type. Respectively, similar effect will be expected at ID value 2^{24} (on the boundary between 3 and 4 bytes) and so on. This behavior repeats when the IDs surpass certain threshold values. However, the increase in the compressed sizes is small enough to be neglected. As explained in this section, for the Full Triple Indexes, each time id_2 changes, the absolute value of id_3 is stored instead of the difference delta. Thus, each time id_1 changes, both id_2 and id_3 are stored as absolute values and need more space. This is why the average item size for the ERT indexes increases from 4.0 to 5.4 bytes for higher absolute ID values.

An upper bound for the compression time can be given with 170 μs. Decompression is for all indexes (except for the ERT index, which is an outlier in this measurement) more than twice as fast as compression and can be lower-bounded by 75 μs. Therefore, the compression and decompression algorithms are fast enough to provide efficient processing for the RFID Triple Store.



(a) ERT Index



(b) ETR Index

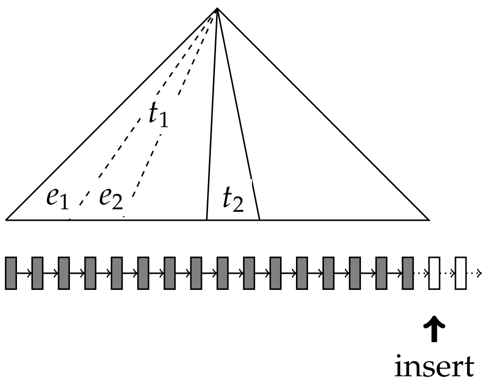
Figure 4.6: E Indexes

4.3 Event Processing

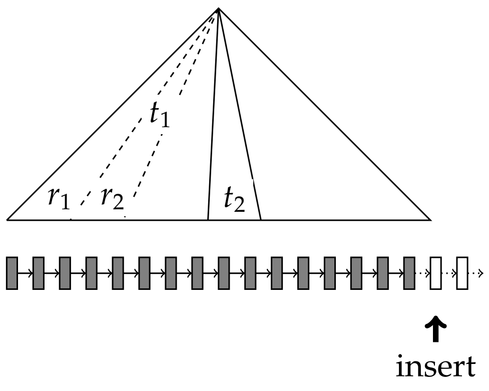
In this section, we provide details about the Triple Store index design that enables a high insert throughput.

4.3.1 Index Design

The 15 indexes differ in how new triples distribute over existing B$^+$-tree leaf pages. The potentially most expensive part during an insert operation is allocating new pages. This operation involves finding free space inside the index segment and extending the segment physically if none is available. Furthermore, the insertion of newly allocated pages destroys the very convenient clustering of the data. In order to support efficient index updates, we therefore reserve (pre-allocate) index leaf pages in advance, which we refer to as *spare pages*. Rather than being allocated on-demand, a range of multiple spare pages is allocated at once. This mechanism has two considerable benefits: (1) it can speed up inserts as it obviates on demand page allocation, (2) it also speeds up sequential scans in queries as spare pages preserve the clustering of the data. Using the data dictionaries, new EPC, reader, and timestamp values will be mapped to unused IDs that are higher than all existing IDs for the corresponding value type. In the indexes, triples are ordered by their IDs, not by their values. Consequently, all (v_1, v_2, v_3) triples with a newly created ID for v_1 will be placed at the right end of the tree. For this purpose we pre-allocate spare pages at the right end of most indexes. In the following, we describe for each index at which position(s) spare pages are needed and how the different indexes manage upcoming events.



(a) TER Index



(b) TRE Index

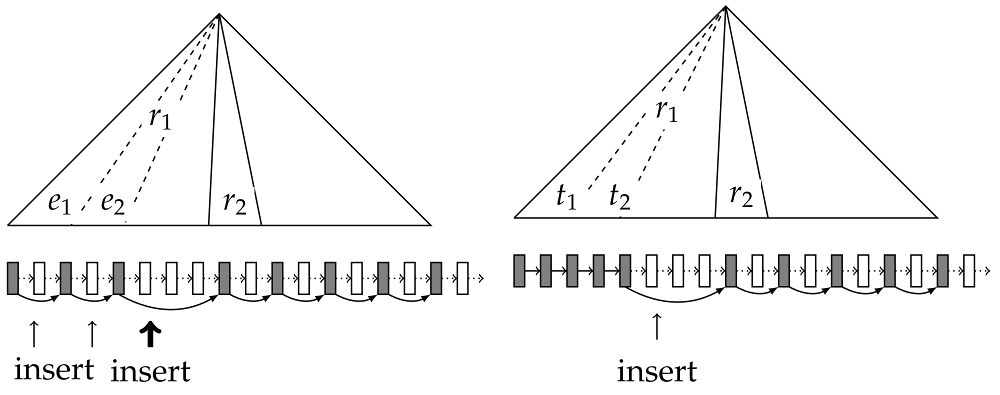
Figure 4.7: T Indexes

E Indexes

Figure 4.6 illustrates the Full Triple Indexes ordered by E first. The sequence of regular, forward-linked leaf pages is shaded gray and spare pages are white. As objects are supposed to enter and leave the sensor infrastructure within a certain time window (corresponding to their production or transportation time), most new events are inserted at the tail of the indexes. This is indicated by the bold arrow denoting the predominant insert position. However, depending on the event generation frequency and the length of the production chain, some inserts might occur in between, as indicated by the thin arrows. For the ERT index, events generated by different sensors, which read an existing EPC e_1 might be inserted at different positions within the index leaves covered by e_1. For instance, e_1 might first be read by r_2, then by r_1. The second event must be inserted before the first as the indexes are sorted lexicographically. Thus, small gaps must be left to fit triples concerning existing EPCs and readers. For the ETR index, it is clear that new events for an existing EPC e_1 can only be appended at the right end of "its" index leaves as past timestamps won't appear again. At the same time, there must be enough space at the tail to hold new objects that did not appear yet.

T Indexes

Figure 4.7 illustrates the Full Triple Indexes ordered by T first. New events are always appended to the tail of the indexes, because recent events have greater (or equal) timestamps than existing events. Therefore, no space at all is required among the existing triples. We only reserve spare pages at the tail of the index tree.



(a) RET Index **(b)** RTE Index

Figure 4.8: R Indexes

R Indexes

Figure 4.8 illustrates the Full Triple Indexes ordered by R first. All sensors in the infrastructure continuously read EPCs and generate events. Thus, the insertions are distributed over the whole breadth of the tree. For a static infrastructure, we do not need to reserve spare pages for newly created reader IDs. For the RET index, most of the inserts will be at the tail of one reader's index leaves. This is because new objects are much more likely to produce events than old objects. Further, the readings of reader r_1 might occur in an arbitrary order, e. g., first e_2 and then e_1. Here again, the lexicographical order requires the second reading to be stored before the first in the index. In the RTE index, new events for a reader r_1 are always appended to the last event within the index leaves covered by r_1.

4.3.2 Analysis of Index Updates

As a first step towards designing the described indexes, we provide a thorough analysis of the number of needed spare pages and their position for each index. We exploit the following facts about traceability data: (1) the sensor infrastructure in a particular application scenario is known in advance and is not supposed to change, (2) the expected event generation frequency for a particular application scenario can be approximated in advance, e. g., due to publications like the BMW experience report [8], as discussed in Section 2.2, (3) timestamps increase monotonically so that younger events have a greater or equal timestamp than older events, (4) monitored objects are supposed to move steadily from reader to reader and disappear from the scene after they have traversed a path of a certain length, e. g., a production process, so that their EPCs are unlikely to appear again in future events. The path length for an object in a particular application scenario is thus predictable.

We determine the number of tuples expected to occur in the next S seconds and

ERT, ETR	Reserve space for $u(v_1) - n[v_1,\_,\_]$ items after each group of $(v_1,\_,\_)$ triples; Reserve space for $f \cdot S$ items at the tail.
ER, ET	Reserve space for $u(v_1) - n[v_1,\_,\_]$ items after each group of $(v_1,\_)$ triples; Reserve space for $f \cdot S$ items at the tail.
E	Reserve space for $f \cdot S/u(v_1)$ items at the tail.
RET, RTE	Reserve space for $w(v_1) \cdot f \cdot S$ items after each v_1-run.
RTE	Reserve space for $w(v_1) \cdot f$ items after each v_1-v_2-run.
RE	Reserve space for $w(v_1) \cdot f \cdot S/u(v_2)$ items after each v_1-run.
RT	Reserve space for S items after each v_1-run.
R	Reserve space for $w(v_1) \cdot f \cdot S$ items after each v_1-run.
TRE, TER	Reserve space for $f \cdot S$ items at the tail.
TR, TE	Reserve space for $f \cdot S$ items at the tail.
T	Reserve space for $f \cdot S$ items at the tail.

Table 4.2: Using the prediction model for calculating the spare pages for each Triple Store index.

estimate the number of spare pages that need to be pre-allocated for each index. We use the notation $n[\cdot,\cdot,\cdot]$ for the number of (e,r,t)-triples with a certain pattern $[\cdot,\cdot,\cdot]$ which are currently (at time t_0) in the database. For example, for the ERT index $n[v_1,\_,\_]$ denotes the number of tuples with EPC v_1. With f we denote the event generation frequency in events per second. The notation $u(e)$ represents the expected number of events generated by a certain object. The variable u is the median of all existing EPC path lengths. The function $w : R \to [0,1]$ denotes for each known reader its relative weight such that $\sum_r(w(r)) = 1$.

Using these parameters, we can calculate the required information in order to "prepare" the spare pages for all expected triples in the time between now (t_0) and $t_0 + S$. Table 4.2 shows a summary of how these considerations are applied for calculating the spare pages of each index. In the ERT and ETR indexes, we need to anticipate $u(v_1)$ events for an EPC value v_1. Considering the number $n[v_1,\_,\_]$ of already stored events for this EPC value, we need to reserve space for another $u(v_1) - n[v_1,\_,\_]$ events after the respective group of $(v_1,\_,\_)$ triples. The same considerations hold for the ER and ET indexes. Most of the events in the E and T indexes are inserted at the tail of the index leaf level. For ease of computation, space for $f \cdot S$ events can be reserved at the end of each index, i.e., the total number of expected events in a time interval of length S. For the E index, we need only reserve space for $f \cdot S/u(v_1)$ events as the $u(v_1)$ expected events for one EPC are aggregated into one value for this index.

For the indexes RET and RTE, we have to reserve space per reader for the expected events. The total number of expected events during the time interval S,

i. e., $f \cdot S$, is thus multiplied by the weight function of the respective reader $w(v_1)$ to determine the number of events generated by reader v_1. For the RTE index, we further split the reserved space into smaller spaces for each v_1-v_2-run. For each reader and timestamp value we expect $f \cdot w(v_1)$ events to be inserted. For the RE index we divide the total number of expected events for the respective reader value by the average path length of an EPC because the path is aggregated into the count value of this index. In the RT index we expect S different timestamp values for a reader as we assume all readers to continuously generate events in our traceability scenario.

For all T indexes, space is reserved only at the right end of the index tree (for $f \cdot S$ triples), because of the monotonically increase of timestamps.

The presented analysis shows how the different indexes behave during updates and determines the number of spare pages needed for a particular scenario. Using these observations, but generalizing our ideas, we present the implementation of the indexes in the next section. By providing auto-tuning mechanisms we automatically adapt the Triple Store to the changes in the environment and omit the need of specifying the predicted parameters $u(e)$ and $w(r)$.

4.3.3 Index Implementation

Providing correct and up-to-date values to the parameters presented in Section 4.3.2 can be inconvenient if not impossible for the database administrator. In the following we describe an approach for dynamic adaption of the RFID Triple Store. The sequence of leaves in Figures 4.6, 4.7, and 4.8 is logically divided into page ranges, called *chunks*. Each chunk consists of a sequence of regular, forward-linked leaves and a range of spare pages. For the R-indexes, one chunk per reader is provided, whereas all other indexes contain one big chunk. As soon as all available spare pages are used up, a chunk grows automatically by a number of pages proportional to its size (we use a grow factor of 50%). All chunks grow independently, and they grow less frequently the larger they get. Therefore, a chunk of a heavily-frequented reader will grow faster and in larger increments than a chunk of an average reader. The leaf page structure will thus eventually adapt to the reader weights. We therefore no longer need the weight function $w(v_1)$ from Table 4.2 to be provided by the database administrator. For the R indexes, this technique successfully adjusts the number of spare pages needed for each reader. For the T indexes, free spare pages are needed only at the tail of the indexes as shown in Figure 4.7 and these are reserved using the automatic growing factor. For the E trees, the spare pages that are reserved at the right end of the tree are determined the same way as for the T indexes. The spare pages that have to be pre-allocated at the positions in between (see Figure 4.6) still use the formulas stated in Table 4.2. However, the expected path length $u(v_1)$ for a new EPC value v_1 can be determined from an existing EPC e' that is known to belong to a similar group of objects (product group) by looking up the *count* value for e' in the Fully-Aggregated E Index. This way, we can dynamically adapt the system to the current load.

Applying the self-adaption of the RFID Triple Store indexes, the leaves of the indexes are filled up on average more than 90 percent and overflows (due to an unexpectedly high number of actually generated events) are rare.

4.3.4 Pre-allocation of Spare Pages

The space allocation mechanism inserts the correct amount of space (the calculated spare pages) at the respective positions on the index leaf page level. The implementation of the pre-allocation is carefully crafted so as to keep the computational overhead minimal. Given an ordered stream of triples (that should be merged with the triples already stored), the algorithm can decide to insert free space by only looking at the currently processed triple and its successor. The decision of whether to leave space behind a triple $\hat{t} = (v_1, v_2, v_3)$ from the input stream depends on $\hat{t}$ itself and the next triple $\hat{t}' = (v'_1, v'_2, v'_3)$ in the stream. All considerations described here are for Full Triple Indexes, but are equally applied to the Aggregated and Fully-Aggregated Indexes. If, for example, $v_1 \neq v'_1$, then $\hat{t}$ is the last triple in a run of triples with equal v_1 values. For the ERT index, this would mean that a new run of events concerning the EPC v'_1 begins behind $\hat{t}$, and space has to be left for future events concerning EPC v_1. In addition to $\hat{t}$ and $\hat{t}'$, the values of three counters are considered: $n[v_1]$, the number of triples in the current v_1-run, $n[v_1, v_2]$, the number of triples in the current v_1-v_2-run, and $n[v_1, v_2, v_3]$, the number of triples in the current v_1-v_2-v_3-run. These counters are incremented or reset based on the values of $\hat{t}$ and $\hat{t}'$ while processing the input triple stream. In the example, after processing $\hat{t}$, their values are all reset to 1. In fact, $n[v_1, v_2, v_3]$ will always be 1 as there cannot be duplicate triples. Note that the counters correspond to certain $n[\cdot, \cdot, \cdot]$ values from Section 4.3.2. For example, for the ERT, ER, ETR, ET, and E indexes, $n[v_1]$ equals $n[v_1, \_, \_]$.

Conceptually, reserved space might be needed at any position where a v_1-run, a v_1-v_2-run, or a v_1-v_2-v_3-run ends, to hold a certain number of expected triples n that belong to the respective run. In other words, any run can "own" reserved space for a number of triples. Usually, many runs fit on a single database page (including reserved space), and it is not convenient to reserve space directly behind a run. Instead, the reserved space for all runs on a page is accumulated and placed behind the existing triples.

A difficulty lies in the fact that a number of tuples, in the presence of leaf compression, may occupy a varying number of bytes when stored on a page. We therefore introduce the average triple sizes ts_o for each index o. The self-adaption mechanism can obtain ts_o at virtually no cost each time a page is compressed. Now, given that space should be reserved for n triples and that another n_{sp} triples fit on the currently processed leaf page P with page size ps, the number of spare

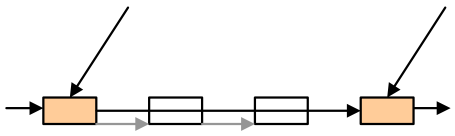


Figure 4.9: Spare pages (adopted from [11]).

pages p to be pre-allocated behind P is:

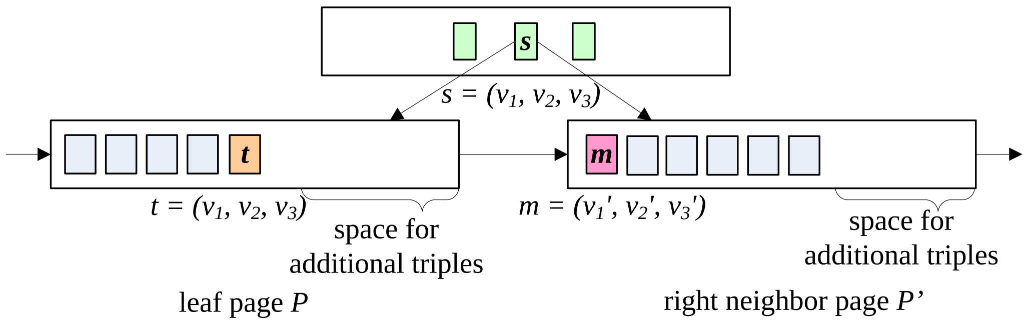
$$p = \begin{cases} 0 & n_{\text{sp}} > n, \\ \lceil (n - n_{\text{sp}}) \cdot ts_o / ps \rceil & \text{otherwise.} \end{cases}$$

A range of spare pages can therefore be assigned to a singe leaf page. In the header of the leaf page, an extra field points to the first available spare page. In Figure 4.9 a leaf page, its two spare pages, and its right neighbor page are depicted. As shown, spare pages are invisible and not linked in the tree structure as long as they are not used. Hence, spare pages are not considered by the lookup algorithm and sequential scans are not affected by them, as they navigate through the leaf pages using their next pointers which always refer to non-empty pages. If a spare page is needed during updates it is only linked in. In case no more spare pages are available, a new empty leaf page has to be allocated.

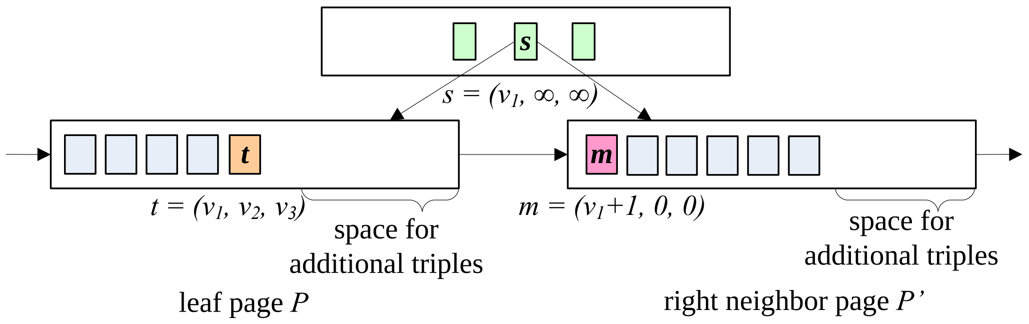
Inner Keys and Merge Limits

In order to correctly reserve spare pages in the index structures of the RFID Triple Store, some implementation details have to be discussed. When new triples are merged in the B$^+$-tree data structure, the following has to be considered: if the particular leaf page P has a right neighbor page P', the left-most key $m = (v'_1, v'_2, v'_3)$ on P' is used as a so-called "merge limit" for those new entries that are merged onto P, as illustrated in Figure 4.10a. This means that all keys merged with entries from P must be smaller than the first key on its neighbor leaf P'. Normally, when the merging on P is processed, the right-most key $t = (v_1, v_2, v_3)$ on P is used as the key for the entry on the parent inner node pointing to P, that means t is used as a splitter key $s = (v_1, v_2, v_3)$. Both the merge-limit m and the selection of t as inner key (or splitter key s) have to be revised for the process of pre-allocating free space on a page.

If we assume that space was reserved on page P for a number of triples of the form $t^* = (v_1, \cdot, \cdot)$, which have to be inserted after t, we will encounter the following problem: the triples t^* would not be stored on P, but on its neighbor page P', because t was used as a splitter key and $t < t^*$. This results in wasting the reserved space on P and occupying space reserved for other triples on P'. To avoid this problem we do not use t as a splitter key, but rather a "virtual" key computed from t and m: e. g., $s = (v_1, \infty, \infty)$. This is shown in Figure 4.10b. As a beneficial side-effect, splitter keys no longer have to be updated every single time



(a) Merge limit and inner keys in a B$^+$-tree .



(b) Adjusted merge limit and inner keys.

Figure 4.10: Merge limit and inner keys in the RFID Triple Store.

the maximum key on a page changes, saving some additional overhead.

Another problem is that if entries are bigger than (v_1, ∞, ∞) but smaller than m, they can get merged onto P together with a run, while actually belonging to P'. We set the merge limit m to $(v_1 + 1, 0, 0)$, in order to solve this issue. Figure 4.10b shows the revised values for t, s, and m.

These considerations apply also for the (Fully-)Aggregated Indexes.

Safety Margins on Leaf Pages

As already described, for some indexes we reserve spare pages only at the tail of the index. The space reserving algorithm decides to ignore the triple sizes ts_o for each index o in some cases if it detects that the number of triples to reserve space for is zero for all runs on a page. As a result, maximally-charged pages are created. This is the case for the TER, TE, TRE, TR, and T indexes, as no triples with old timestamps are expected in the stream of newly arriving events. These maximally-charged pages combined with certain specifics of the prefix compression can lead to the following problem: inserting new items on a page could change the compressed sizes of existing triples that are placed after the newly inserted ones. Besides, all aggregated triples apply an extra small encoding if the *count* value of an aggregated triple is less or equal to 4. So, the size of the triples

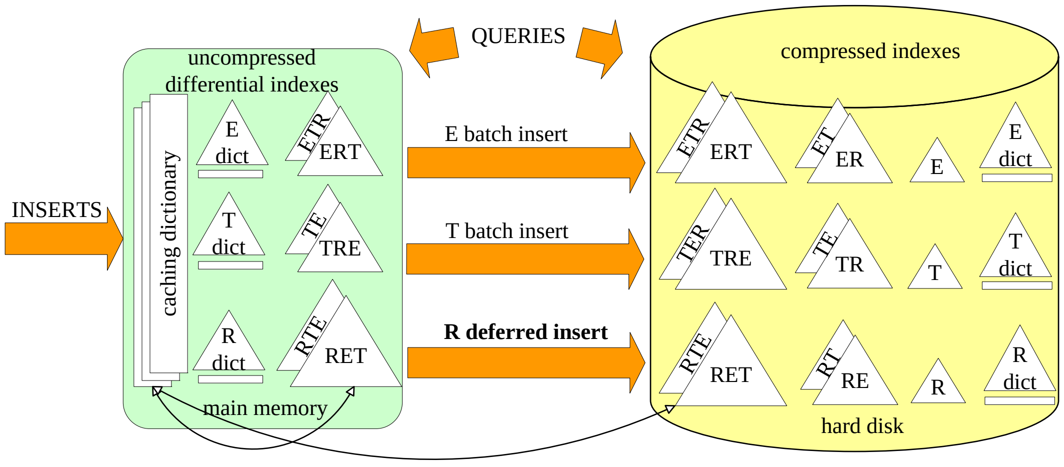


Figure 4.11: Overview of the event processing in the RFID Triple Store.

may slightly grow if the *count* values get updated and exceed 4. As a result, especially maximally-charged pages are likely to overflow, generating undesirable split pages. Therefore, the RFID Triple Store leaves a safety margin of 100 bytes per page. Triples that are packed on a newly created page may not use the safety margin, but on a later update, the space taken by the elements can grow into the safety margin. The 100 bytes take less than 1% of the available page size, but significantly reduce undesirable overflows of the maximally-charged pages.

4.3.5 Index Update

It is a challenge to design the indexes in a way which allows for an efficient update for the heavy continuous insert stream typical for traceability data. In our system, inserts are always done in a batched manner. This means that events are collected in a batch within the interval of one second and are then fired to the system. Figure 4.11 illustrates how the event processing in the RFID Triple Store looks like and gives an overview of the indexes in main memory and on disk. All new triples are first loaded into small differential indexes, which are uncompressed indexes in main memory. In memory only the six differential Full Triple Indexes (ERT, ETR, RTE, RET, TER, TRE) exist. The differential Aggregated Triple Indexes and Fully Aggregated Triple Indexes can be derived from the Full Triple Indexes on the fly, if a query requires them, and are not created in advance.

Periodically, the differential indexes are merged (as batches) with the main indexes on disk. The merging process for the Full Triple Indexes is depicted in Algorithm 7. For each batch, the input is first sorted in the order of the corresponding index (lines 1-5 for RTE, 6-10 for RET, and so forth). Second, for each element of the batch we look up the leaf page on which it belongs and memorize the path for further lookups (see Algorithm 8). Third, we process the input, i. e.,

Algorithm 7: Algorithm processBatch

```
input : batch, bs (batch size)
/* Updating the Full Triple Indexes for one batch       */
/* RTE                                                   */
1  sort batch in RTE order;
2  forall the $r_i$ in batch do
       // 1000 (= $|R|$) times;
3      $p \leftarrow search(r_i, RTE)$;                  /* Algorithm 8 */
4      $process(p)$;                                     /* Algorithm 9 */
5  end
/* RET                                                   */
6  sort batch in RET order;
7  forall the $r_i, e_j$ in batch do
       // 2500 (= $bs$) times;
8      $p \leftarrow search(< r_i, e_j >, RET)$;         /* Algorithm 8 */
9      $process(p)$;                                     /* Algorithm 9 */
10 end
/* TER                                                   */
11 sort batch in TER order;
12 $p \leftarrow$ last not-full page of index;
13 $process(p)$;                                         /* Algorithm 9 */
/* TRE                                                   */
14 sort batch in TRE order;
15 $p \leftarrow$ last not-full page of index;
16 $process(p)$;                                         /* Algorithm 9 */
/* ERT                                                   */
17 sort batch in ERT order;
18 forall the $e_i, r_j$ in batch do
       // 2500 (= $bs$) times;
19     $p \leftarrow search(< e_i, r_j >, ERT)$;         /* Algorithm 8 */
20     $process(p)$;                                     /* Algorithm 9 */
21 end
/* ETR                                                   */
22 sort batch in ETR order;
23 forall the $e_i$ in batch do
       // 2500 (= $bs$) times;
24     $p \leftarrow search(e_i, ETR)$;                  /* Algorithm 8 */
25     $process(p)$;                                     /* Algorithm 9 */
26 end
```

after decompressing the page found during the search, we merge the batch entries with the entries on this page. If the page overflows we can use one of the spare pages. At the end, we write back the resulting triple stream and compress the pages again. If a spare page was used, the next pointer to this page has to be activated. Algorithm 9 describes the processing of the batch items. As we can see, the costs of the merging process depend on the costs for the lookup operation as well as the costs for the compression and decompression of a page.

Algorithm 8: Algorithm search

```
input  : Value $v_i$ or value pair $< v_i, v_j >$, index $i$
output: The first leaf page for $v_i$ or $< v_i, v_j >$ in the specified index
1 $p \leftarrow$root node;
  /* search for insert page(s), first in the differential,
     than in the main dictionaries                          */
2 while $p$ is inner node do
      // 2 (=height of index without root page) times;
3     read page $p$;
4     do a binary search;
5     $p \leftarrow$ reference of next page;
6 end
  /* $p$ is a leaf node                                     */
7 return $p$;
```

Algorithm 9: Algorithm process

```
input  : Page $p$
1 read $p$ (and possibly subsequent pages);
2 decompress $p$;
3 find insert position at end of $p$;
4 compress batch on $p$ (and possibly on subsequent pages);
5 write all updated pages;
```

For the E and T indexes, the triples in a batch are inserted mostly at the tail of the tree, whereas R indexes are updated at each reader's position. For the processing of one batch this requires one lookup per reader to determine the right insert position. Since this is very costly, especially when the indexes are growing in depth, we are following a "hybrid" approach: The four E and T differential indexes are flushed to disk immediately after a batch of triples has been inserted; for the two R differential indexes, on the other hand, we use a staging architecture with deferred index updates and these are flushed to disk much less frequently to avoid high merge costs. Since the T and E differential indexes are merged with the main indexes after each event batch, they are depicted smaller than the R indexes in Figure 4.11. At particular points in time (or when memory is exhausted) the

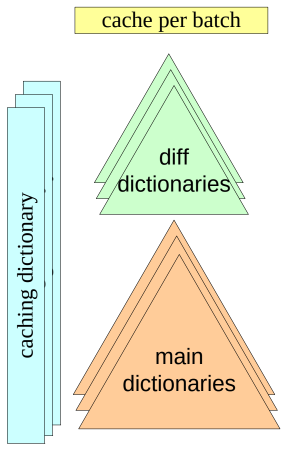


Figure 4.12: Overview of the dictionary and caching mechanisms of the RFID Triple Store.

differential R indexes are flushed into the corresponding B$^+$-trees in a bulk insert operation.

Caching Techniques

Analogously to the differential indexes, three small differential dictionaries (for the E, R, and T values) reside in main memory during a batch execution. This way, dictionary lookups during the batch processing are cached and can be retrieved at no disk I/O cost. Like the other indexes, the delta dictionaries are flushed to disk at the end of a batch execution. Since in an RFID scenario we are mainly querying the recent activities of the objects, we use an additional data structure called a caching dictionary. The caching dictionary keeps current object IDs *across batch boundaries* in main memory. It is organized as a least-recently-used-queue. This caching of dictionary lookups is beneficial to inserts and queries over recent events, which are likely to still reside in the cache. We use three caching dictionaries for the E, R, and T values. In Figure 4.11, an example of caching the last accessed values from the differential and main RET index is depicted by the thin arrows. In Figure 4.12, the dictionary and caching mechanisms of the RFID Triple Store are summarized. Next to the differential and main dictionaries that were already discussed, there exists a very small cache per batch for the repeated information during batch execution, for instance the long reader identifiers. The caching dictionary holds, as already explained, frequently used object IDs across batch boundaries in main memory. These IDs can originate from the differential as well as from the main dictionaries.

```
SELECT o1.e
FROM events o1
WHERE o1.r IN
   (SELECT o2.r
    FROM events o2
    WHERE o2.e=1 AND abs(o1.t − o2.t) < 300)
```

Figure 4.13: Contamination query.

4.4 Query Processing

Business traceability applications need to refer to up-to-date information in order to provide meaningful information for the decision-making processes relying on them. Therefore, these applications have to process not only historical (OLAP) data, but need to additionally involve the latest OLTP information in the analysis process. The RFID Triple Store answers OLTP queries (e. g., "Where is object *o* at this moment?") "up-to-the-second", i. e., they are fast enough in order to provide the information of the last second. Further, the Triple Store supports efficient OLAP query processing, particularly range queries over a time interval, which are typical in RFID applications.

For query processing, the information in the differential indexes *and* the information in the main indexes have to be taken into account. To obtain a complete/unified view on this data, additional union/merge joins (merge joins with union semantics) between the differential indexes and the main indexes are integrated in the query plan during query processing. This is however transparent for the application or user. If there are no relevant or no new triples in the differential indexes during query execution, additional joins with the differential index will be unnecessary. Consequently, the union/merge joins are omitted if the query plan generator detects that a differential index contains no triples that are relevant to a current query. Since we propagate the changes in the T and E indexes immediately (after each batch) to the main indexes, the T and E differential indexes will be empty most of the time. Therefore we don't have the overhead of the corresponding differential index scans or union/merge joins. For the R indexes, we take this overhead into account for the sake of the insert efficiency.

Using some typical RFID queries as an example, we will explain the query processing features of the RFID Triple Store. Figure 4.13 shows an example OLAP query (a contamination query), which determines all objects that passed the same readers as the object with EPC 1 within a time interval of 300 seconds. Its execution plan, which uses only the main indexes, is shown in Figure 4.14. The query parser derives triple patterns of the form (e, r, t). The components of these patterns can be either a variable or a constant value. For our example query, there are two triple patterns: (r_1, e_1, t_1) and (1, r_2, t_2).

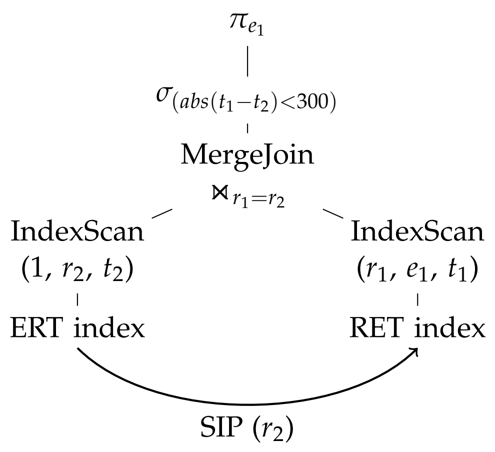


Figure 4.14: Execution plan of the query in Figure 4.13

In the following, we discuss the query engine characteristics of the RFID Triple Store which speed up the query processing.

4.4.1 Index Range Scans and Merge Joins

The thorough indexing of the data and the fact that the data is sorted by all possible column permutations allows most queries to be answered by using a range scan on a specific index. Furthermore, the wide choice of different sort orderings of the indexes enables most often the execution of efficient merge joins. The optimizer plans are geared to preserve important orders in intermediate results for subsequent joins. When this is no longer possible, the query engine switches to hash-based join processing. For our example query in Figure 4.13, the ERT index is used to determine all tuples containing $e = 1$ and a range scan over the index leaves implying (only cheap) sequential I/O is done. Since the result of the left index scan is sorted by R, an appropriate merge join with the RET index on the reader attribute is then executed.

4.4.2 SIP

The query engine of the RFID Triple Store takes advantage of the *sideways information passing* (SIP) technique as described in [39]. This is a mechanism of passing filter information across pipelined operators that process comparable identifiers, in order to accelerate joins and index scans. This information is used to identify irrelevant index parts that will be skipped by parallel operator(s) and thus make query processing more efficient. Our example query also benefits from the SIP technique: the intermediate results produced by scanning the ERT index for $e = 1$ are passed to the RET scan, which will therefore skip irrelevant index parts by jumping directly to the "correct" reader values. Since in our scenario an EPC is in

```
SELECT DISTINCT e
FROM events
WHERE r = 1
  AND ts > 5
  AND ts < 10
```

Figure 4.15: Range query: A list of objects scanned by sensor 1 within a specified time interval.

average scanned by 20 out of 1000 readers, this will result in only accessing 2% of the RET index leaves.

As described in [39], the SIP technique is only applied to the main indexes, since the differential indexes are comparatively small. We, however use the deferred index update for the R indexes as described in Section 4.3.5 and keep the R indexes for as long as possible in main memory, so that they can grow large. For this reason, we also implement the SIP technique for the R differential indexes, which brings a significant performance benefit for the queries that use these indexes (see Section 4.5).

4.4.3 Order-preserving Dictionary

As mentioned in Section 4.2, using three dictionaries for each of the triple columns E, R, and T enables considerable query processing optimizations. Due to the fact that the timestamps of scanned events increase monotonically, the ID-mapping in the T dictionary is order-preserving. Furthermore, through the employment of dedicated data dictionaries, the assigned dictionary IDs are contiguous. We leverage this characteristic in query processing and apply three different optimization methods: (1) IDs-only comparison, (2) pushing down selections, and (3) extended SIP.

IDs-only Comparison

Most of the queries in an RFID scenario are time-restricted, i. e., they are usually concerning a specific time interval or are interested in events that occurred at a particular time. We consider queries containing selection patterns like $t < value$ or $t_1 < t_2$ as shown in Figure 4.15. Observing the query plan for this range query in Figure 4.16, we can see that after performing the RET index scan, the selection predicates on the timestamp values are evaluated. One will usually first perform dictionary lookups in order to determine for each tuple (in this case the result coming from the scan) the actual value corresponding to the dictionary ID t.

Only then this actual value (say t_1) can be finally compared to the given time value or to another determined timestamp value (t_2). Taking advantage of the order-preserving characteristic of the T dictionary, we can omit the dictionary

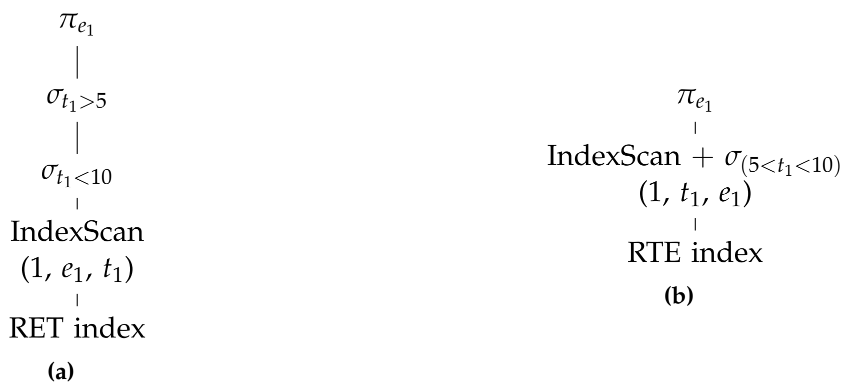


Figure 4.16: Execution plan of the query in Figure 4.15 before and after optimization

lookups in this case and perform the comparison directly on the ID values. For the first pattern ($t < value$) we determine the ID of the given value once at query compilation time and then compare it to the timestamp IDs t. For the second pattern ($t_1 < t_2$) we just compare the IDs of values t_1 and t_2. Query execution is drastically accelerated by this technique (by several orders of magnitude).

Pushing Down Selections

Further, for range queries over a time interval, we can push down the selection in the index scan. Consider the example query in Figure 4.15. Since we know the timestamp IDs of the given values (5 and 10), we can choose an index scan where the input is sorted by T and can only iterate over the tuples in the applicable time range. In Figures 4.16a and 4.16b, the original query plan and the query plan after pushing down the selection is shown. By choosing the RTE index, the input is sorted by T since the reader value is set to a particular value provided by the query ($r = 1$ in this case). Thus, during the scan, we can only iterate through the tuples where the timestamp is bigger than 5 and smaller than 10. It is, however, not always possible to choose an index scan that is sorted by T. For example if a particular ordering is needed for building a merge join, then this ordering is preferred. This is due to the fact that fast merge joins are always favored over hash joins. We extended the existing cost model in the query engine of the RFID Triple Store by the special case of pushing down selection in range queries.

The decision which index scan should be preferred is based on the expected output cardinality of this scan, i. e., how many pages/items have to be read. If the selection predicate of a range query can be pushed down, i. e., if the selection predicate is time restricted, we integrate start and stop conditions within the index scan. Thus, the expected output cardinality of the index scan is adjusted for this case and this optimized plan will be favored. However, each operator calculates its own costs so that even if the index scan with the optimized variant of pushing

```
SELECT o1.e
FROM events o1, events o2
WHERE o1.e = o2.e
  AND o1.r = 1
  AND o2.r = 2
  AND o1.ts > 5
  AND o2.ts < 10
```

Figure 4.17: Range query: A list of objects, which were scanned by sensor 1 after a time threshold and by sensor 2 before a time threshold. Consider, that the sensors have a specific semantic, like entry and exit. This query determines for example the objects that passed an entry sensor after 8 o'clock in the morning and an exit sensor before 12 o'clock in the morning, i. e., all objects produced before noon.

down the selection would have been possible, another index ordering can be chosen. Especially if that will result in applying a more efficient merge join, rather than hash join in the query processing as can be seen in Figure 4.17. For these cases, where for range queries an index ordering different from T was chosen, we can again apply an optimization within the index scan. We prune the result set and ignore the irrelevant parts of the index scan, i. e., skip them, as shown in Figure 4.18. For each of the RET index scans, we pass only the relevant tuples: $t_1 > 5$ and $t_2 < 10$ respectively. By omitting the irrelevant triples during the scan, we pass less triples to the next operators, and thus increase the performance of the query processing. This optimization brings a considerable performance benefit for range queries (up to one order of magnitude), which are typical in an RFID scenario. For more details, see Section 4.5.

Extended SIP

Consider again our example query plan in Figure 4.14. As explained in the beginning of the section, this query benefits from SIP, since the right index scan takes advantage of the r_2 hints of the left index scan and can directly jump to the leaf pages containing the reader values. Instead of producing the intermediate results of the merge join and applying the selection predicate on top of it, we could also pass the corresponding t_2 values along with the r_2 values using the SIP technique and push the selection down this way. This is again possible due to the order-preserving T dictionary. We can perform all comparisons directly on the timestamp ID values. Further, if we apply these two different SIP hints (r_2 and t_2) on the index scan over RTE in place of RET, we will just have to jump to each of the r_2 hint values and iterate over the T column as long as the condition $t_2 > t_1 - 300$ holds (assumed that $t_1 > t_2$). For queries that benefit from the SIP technique anyway and which also contain a selection predicate over timestamps,

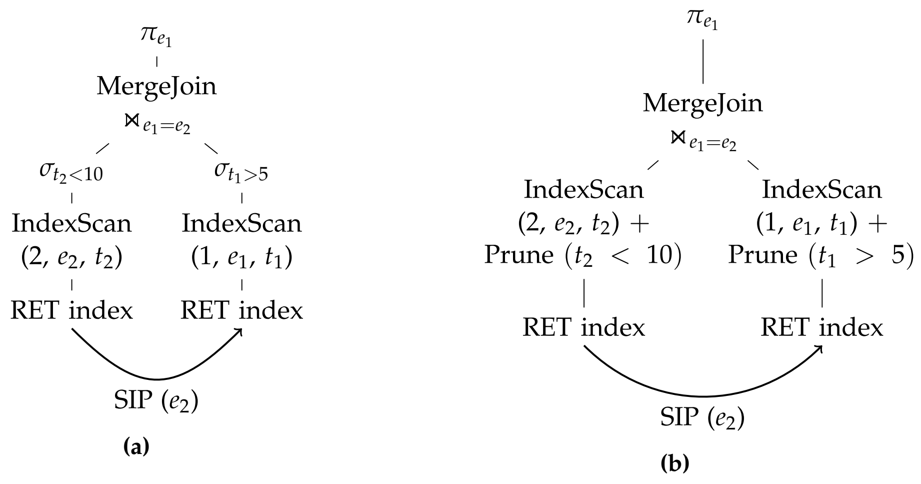


Figure 4.18: Execution plan of the query in Figure 4.17 before and after optimization.

this smart optimization brings a considerable performance improvement. For the query in Figure 4.14, this technique accelerates the query performance by a factor of 2. For more details, see Section 4.5.

Another implementation of an efficient dictionary-based, order-preserving compression is discussed in [7]. In contrast to that approach, we don't deal with dictionary updates and thus benefit from a very efficient implementation of the data dictionary.

4.5 Performance Evaluation and Comparison

In this section, we present the measurements and analysis of the insert and query performance of the RFID Triple Store. For comparison, we implemented the triple store schema (with the columns E, R, T) using a commercial, general-purpose row-store database system, referred to as DBMS-X, and the column-store open-source database system MonetDB [1, 9].

4.5.1 Experimental Setup

For all experiments, we first pre-load a certain amount of data (The data generated during one week, two weeks and one month with the event frequency of 2500 events per second respectively.) into the database. Then, we execute a mixed workload consisting of continuous inserts, OLTP and OLAP queries running concurrently. We set the insert frequency to 2500 events per second in order to prove the applicability of the particular system for the scenario reported by BMW [8]. As a *measurement of the system's performance*, we evaluate whether the different approaches can maintain the consistent insert throughput of 2500 events per second and at the same time achieve reasonable query response times. We further investigate the scalability of our approach by experimenting with different pre-loaded data sizes.

For the RFID Triple Store and MonetDB there are no configuration parameters to be set. For DBMS-X, the memory parameters were set to be self-tuned. After determining the self-tuned values once, this property was switched off to avoid the effect of back-and-forth tuning. Additionally, we consulted the physical design tuning tool of DBMS-X for index suggestions using our experimental workload as input. The MonetDB system does not support indexing. The remaining approaches use up-to-date indexes and statistics.

All experiments were conducted on a 64 bit-Red Hat Enterprise Linux server with four quad-core Intel Xeon 2.40 GHz CPUs, 16 GB main memory, using one 1 TB SATA disk without RAID.

4.5.2 Data Generation

As there is no publicly available dataset for traceability data, we use an event generator whose implementation follows the RFID data generator implementation provided by IBM [5], extending it by a more flexible object movement. We generate our data based on a realistic RFID scenario to simulate the movement of items through different locations. The created objects are scanned on average by 20 sensors during their lifetime. Ca. 5% of the events represent new objects, and 95% are positional updates of existing objects. Without loss of generality equal distribution of the readers in the infrastructure is assumed. Therefore, with x readers, on average $1/x$ of the events are generated by an individual reader. We use this fact and generate the events at random locations, so that cycles in the movement graph may occur. In our scenario, events are tracked at 1000 different locations. The average path length of the tracked objects is in the range of 19 to 21. This is a realistic life cycle for an object in the traceability scenario. For 10000 batches and 250000 objects the distribution of the path lengths for the objects appearing in the dataset is illustrated in Figure 4.19.

Our event generator is designed as follows: We generate three different datasets for pre-loading – a one-week-dataset, a two-weeks-dataset and a one-month-dataset – in order to analyze the scaling ability of the system. The datasets result in 450 million, 900 million, and 1980 million events respectively, which is significantly bigger than the largest datasets used in the referenced papers [27], [35] and [37]. The database size for the RFID Tripe Store amounts 39 GB, 74 GB, and 170 GB for the different pre-loaded datasets, respectively.

4.5.3 Query Working Set

We create a typical query workload for RFID traceability data consisting of the 11 query types shown in Table 4.3. These queries resemble the queries presented

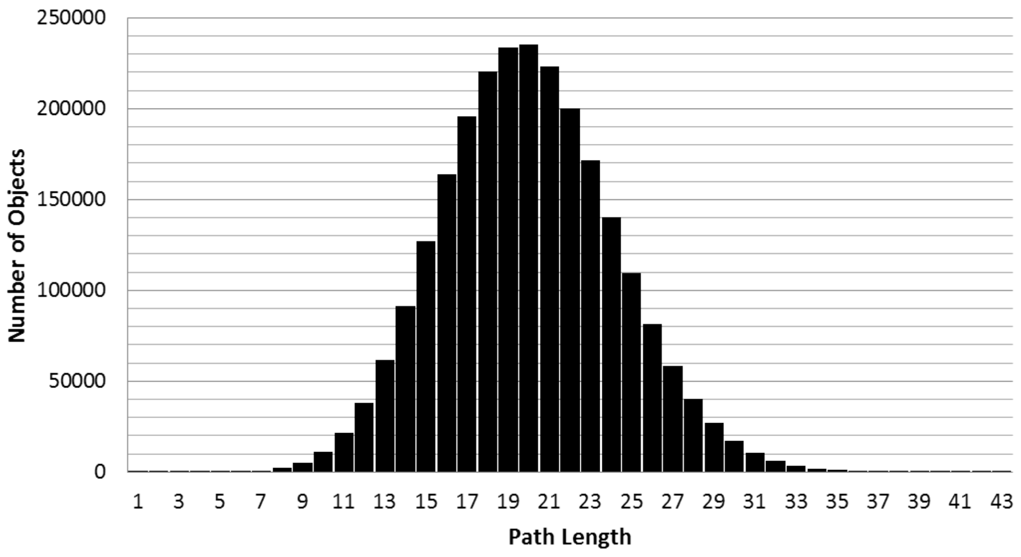


Figure 4.19: Distribution of the path lengths of an object in a test dataset (adopted from [11]).

in Table 3.3. However, some of the queries have been further refined, i. e., constrained by a time interval. For this reason we provide again a short description of the query workload. Q1 and Q2 are OLTP-style queries that are supposed to be submitted for every object in the database. Q1 determines the last position of an object and Q2 the pedigree of an object [5]. This type of queries is common in scenarios like parcel services: e. g., tracking a packet. The remaining 9 queries are OLAP-style queries. Q3 and Q4 collect information about a certain sensor. Q3 determines the number of objects scanned by the sensor and Q4 lists all items that have been scanned by the sensor in a specific time interval. This information is important in production scenarios in order to detect if a certain sensor is overloaded or if there is a bottleneck in the production process. Q5 through Q7 collect data about items that were scanned by multiple (in this case two) certain sensors. Q5 lists all objects that have been seen by two different sensors within a particular time interval. Q6 focuses only on objects that have crossed two consecutive sensors within a time interval and Q7 counts the objects determined by Q6. The results might provide diagnostic information about a certain possibly faulty route of transport. Furthermore, Q6 and Q7 show the difference between only counting the triples and using the reverse lookup in order to map the IDs to literals as described in Section 4.2.2. Q8 is a *contamination query*, as described in detail in Section 4.4. It determines all objects that were with a certain object at a certain sensor within a specified time interval. If, for example, an error occurs at one station of the production chain so that all goods being at the same place are contaminated, this query provides all possibly affected objects. Q9 through Q11 provide an overview of statistical information about one or multiple sensors, e. g., for determining production peaks. Further, information about particular object routes or specific paths is gathered. This kind of information is usually required for regular reports, e. g., for providing an overview of the whole infrastructure and its load. Thus, these are OLAP-style queries, processing big amounts of data and usually long-running. OLAP queries are typically submitted for report generation or decision making and occur less often than OLTP queries.

4.5.4 Mixed Workload

No.	Query
Q_1	Last location of an object
Q_2	The pedigree (complete path) of an object
Q_3	The number of objects scanned by a certain sensor
Q_4	A list of objects scanned by a sensor within a time interval
Q_5	A list of objects, which were scanned by two sensors s1 and s2 within the given time constraints
Q_6	A list of objects, which were scanned first by sensor s1 and then by sensor s2 within a time interval
Q_7	The number of objects, which were scanned first by sensor s1 and then by sensor s2 within a time interval
Q_8	A list of objects that were at sensor s, together with an object x (suspected to be contaminated) within a certain time interval (contamination query)
Q_9	Listing the number of objects per reader and timestamp which passed in a certain time interval (e.g., to identify production peaks)
Q_{10}	Listing the number of all objects scanned by all 10 readers within a time interval, grouped and ordered by reader, and a time interval of a second
Q_{11}	Listing the number of all objects which were scanned by the sensors s1, s2, and s3 within a time interval in this order, aggregated per second

Table 4.3: Query workload for a typical RFID traceability scenario.

For the mixed workload, inserts and queries (OLTP and OLAP) are executed concurrently in the system. We are interested in (1) whether the pre-defined throughput can be managed by the approaches, and (2) how the query response times behave during the benchmark. The mixed workload is designed as follows: One insert client constantly inserts 2500 events per second over the course of 10 hours into the database. This simulates the load (event generation) produced during one working day. Concurrently, two query clients are running in the system: one OLTP and one OLAP client. The OLTP client sends one of its two queries (Q1 and Q2) and after that simulates a think time of 10 ms (i. e., almost every new object is queried once). The OLAP client executes one of its 9 queries and has a think time of 1 s.

Mixed workload

Insert TP	No.	QRT [ms] 1 week		QRT [ms] 2 weeks		QRT [ms] 1 month	
		avg	95th perc	avg	95th perc	avg	95th perc
2500 events/s	Q1	1	2	3	21	19	85
	Q2	1	2	3	21	20	85
	Q3	1	1	1	1	2	2
	Q4	39	52	84	93	263	369
	Q5	33	37	62	69	265	268
	Q6	40	38	62	70	155	274
	Q7	11	12	21	22	58	172
	Q8	1432	1872	2868	3838	8905	13474
	Q9	43	24	77	24	303	79
	Q10	15	9	23	9	15	43
	Q11	17	28	55	56	110	288

Figure 4.20: A mixed workload of concurrent inserts and queries on the RFID Triple Store. The query performance for the three different pre-loaded datasets is shown: one week, two weeks and one month. The system copes with the sustained insert throughput of 2500 events per second. The query response time (QRT) for each setting, divided in average response time (avg) and the 95th percentile (95th perc) is depicted.

RFID Triple Store

For our approach we pre-load the three different datasets (one-week-dataset, two-weeks-dataset and the one-month-dataset) and conduct the mixed workload experiment for each of these settings. As explained in Section 4.3.5, the T and E indexes are flushed to disk immediately after each batch, whereas for the R indexes we perform a deferred update. The differential R indexes are incrementally merged with the indexes on disk, i. e., the main indexes are not reconstructed from scratch each time, but only the new differential parts of the index are incrementally added. For our use case, we experimentally chose to perform the merge each 3000 batches (each batch contains 2500 events). The costs for the merging are increasing only slightly for bigger database sizes: the margin between the one-week-dataset and the one-month-dataset ranges from 75 seconds to 120 seconds. But this extra overhead for merging, resulting in a number of unprocessed batches is quickly compensated by the insert client, which executes the overdue batches without delay.

For all three experiments, the RFID Triple Store achieved the desired sustained throughput of 2500 events per second. The query performance for each data size can be seen in Figure 4.20 and is discussed in comparison with the other approaches below.

Comparison of the Approaches

Mixed workload				Query-only workload	
Insert TP	No.	QRT [ms]		QRT [ms]	
		avg	95th perc	avg	95th perc
244 events/s	Q1	9697	38219	17	77
	Q2	9774	36658	23	95
	Q3	7058	13777	285	590
	Q4	6668	14359	35	97
	Q5	10249	20210	799	1783
	Q6	9013	20546	607	1587
	Q7	12065	23869	4583	9194
	Q8	50918	76618	24712	32826
	Q9	6323	12816	121	245
	Q10	17254	26947	9024	10790
	Q11	19656	35107	7481	11412

Figure 4.21: A mixed workload of concurrent inserts and queries and a query-only workload on the one-week-dataset for DBMS-X.

For the comparison experiments, we take the one-week-dataset, which results in the following database sizes including indexes: 39 GB for the Triple Store, 345 GB for DBMS-X, and 35 GB for MonetDB. We conduct two experiments with DBMS-X: (1) a mixed experiment with inserts and queries and (2) a query-only experiment, where only the two query clients (OLTP and OLAP) are running on the initially pre-loaded data. For MonetDB, we perform only the query experiment, i. e., (2), since this system is not optimized for heavy insert workloads as stated in [1].

The performance results of DBMS-X and MonetDB are shown in Figures 4.21 and 4.22, respectively. In general, MonetDB performs poorly here, DBMS-X better and the RFID Triple Store is the best among the three approaches. When con-

Query-only workload		
No.	QRT [ms]	
	avg	95th perc
Q1	$2386 \cdot 10^3$	$4182 \cdot 10^3$
Q2	$3297 \cdot 10^3$	$7031 \cdot 10^3$
Q3	$3589 \cdot 10^3$	$4189 \cdot 10^3$
Q4	15798	26564
Q5	$6472 \cdot 10^3$	$7055 \cdot 10^3$
Q6	$4947 \cdot 10^3$	$5588 \cdot 10^3$
Q7	$12958 \cdot 10^3$	$13300 \cdot 10^3$
Q8	$2971 \cdot 10^3$	$3136 \cdot 10^3$
Q9	12544	13571
Q10	$365 \cdot 10^3$	$428 \cdot 10^3$
Q11	$13145 \cdot 10^3$	$17528 \cdot 10^3$

Figure 4.22: Query-only workload on MonetDB.

ducting a mixed experiment with inserts and queries on DBMS-X, the system only achieves an insert throughput of 244 events per second. The average query response times here are up to three orders of magnitude higher than those for the Triple Store. The concurrent inserts that have to touch and update all the indexes affect greatly the query performance of DBMS-X. We can verify this in the query-only experiment for DBMS-X, where the query response time improves by up to two orders of magnitude. The MonetDB performance suffers from flushing data to disk (lacking memory) and the absence of indexes for the OLTP queries. Q4 and Q9 benefit from the column-wise storage and perform better than the rest of the queries on this system.

The corresponding indexes used by DBMS-X and the Triple Store, which are listed in Figure 4.23 are almost identical for each query. This means that the physical design tool of DBMS-X proposed the same indexes as used by the Triple Store on the given workload. However, the indexes of DBMS-X are not compressed and their size is considerably bigger than the size of the Triple Indexes. DBMS-X's query performance for the query-only workload is in general by one order of magnitude worse than the mixed query performance of the Triple Store. Queries Q1 and Q2 can be expected to be very frequent in OLTP-tracing applications, so it is particularly important that they are executed very efficiently. The RFID Triple Store significantly outperforms DBMS-X for these two queries. Both systems make use of the ETR index to determine the location(s) for a particular EPC. However, this index is very large for DBMS-X, it consumes more disk space than the event table itself. Consequently, the Triple Store reads less data due to its highly com-

Used Indexes			
Query No.	DBMS-X	Triple Store	SIP
Q1	ETR	ETR	-
Q2	ETR	ETR	-
Q3	R	R	-
Q4	RTE	RET	-
Q5	RTE, RET	2 x RET	SIP
Q6	RTE, ERT	2 x RET	SIP
Q7	2 x RET	2 x RET	SIP
Q8	ERT, RTE	ERT, RET	SIP
Q9	TR	TR	-
Q10	RT	TR	-
Q11	3 x RET	3 x RET	SIP

Figure 4.23: Indexes used by the queries executed on the DBMS-X and the RFID Triple Store. In the last table column queries that benefit from the SIP technique of the Triple Store are shown.

pressed indexes, which explains its performance gain for these queries. Query Q3 can be answered extremely efficiently by the RFID Triple Store due to the Fully Aggregated R index. This index contains a count value for each reader, which is read and returned by the query. Further, the Fully Aggregated Index R is highly compressed and therefore fits into main memory. Queries considering time intervals (range queries) or relations between timestamps, such as Q5, Q6, Q7, Q9, Q11 leverage the order-preserving dictionary in sorts and comparisons of timestamps as described in Section 4.4. Except for Q9, these queries additionally benefit from the SIP technique used in the RFID Triple Store. For query Q4, both systems have a similar performance. DBMS-X exploits the RTE index for this query, whereas the Triple Store scans over all qualifying RE-triples first and applies the selection on top (having a bigger intermediate result). Q8 is a contamination query, whose execution plan is discussed in Section 4.4 (see Figure 4.14). It also benefits strongly from the SIP processing. Even though both systems require the TR index for Q9, these two indexes differ greatly in their implementation. The Triple Store can take advantage of the count values and doesn't need to perform aggregation first, whereas DBMS-X does. The Triple Store takes the same advantage for Q10 that also uses the TR index.

Mixed workload					
Query No.	Query RT [ms] 1 week		Query RT [ms] 1 week optimized		
	avg	95th percentile	avg	95th percentile	card
Q1	1	2	1	5	1
Q2	1	2	1	5	19
Q3	1	1	1	1	count 494966
Q4	39	52	2	3	29
Q5	33	37	12	15	83
Q6	40	38	11	13	29
Q7	11	12	11	12	count 29
Q8	1432	1872	612	1161	23
Q9	43	24	39	23	29900
Q10	15	9	4	5	286
Q11	17	28	17	28	2

Figure 4.24: A mixed workload of concurrent inserts and queries on the RFID Triple Store (one-week-dataset). Query response time after the optimizations.

Mixed workload					
Query No.	Query RT [ms] 2 weeks		Query RT [ms] 2 weeks optimized		
	avg	95th percentile	avg	95th percentile	card
Q1	3	21	3	24	1
Q2	3	21	4	24	19
Q3	1	1	1	1	count 945450
Q4	84	93	4	16	29
Q5	62	69	25	35	59
Q6	62	70	23	28	29
Q7	21	22	21	22	29
Q8	2868	3838	1173	2370	23
Q9	77	24	20	23	29900
Q10	23	9	4	5	287
Q11	55	56	55	56	2

Figure 4.25: A mixed workload of concurrent inserts and queries on the RFID Triple Store (two-weeks-dataset). Query response time after the optimizations.

Mixed workload					
Query No.	Query RT [ms] 1 month		Query RT [ms] 1 month optimized		
	avg	95th percentile	avg	95th percentile	card
Q1	19	85	15	57	1
Q2	20	85	15	57	19
Q3	2	2	2	2	count 1844343
Q4	263	369	59	104	29
Q5	265	268	110	305	59
Q6	155	274	84	241	29
Q7	58	172	58	172	29
Q8	8905	13474	3927	8206	23
Q9	303	79	25	54	29900
Q10	15	43	36	7	71197
Q11	110	288	110	288	2

Figure 4.26: A mixed workload of concurrent inserts and queries on the RFID Triple Store (one-month-dataset). Query response time after the optimizations.

Query Optimizations

The performance numbers presented in Figure 4.20 are based on the optimizations discussed in the first paragraph of Section 4.4 (IDs-only comparison). By working only on IDs for the T values, the query execution is accelerated by several orders of magnitude. This was the first and most significant query speed up for our RFID Triple Store. As already mentioned, range queries which select a particular time interval are very typical in a traceability scenario, such that the IDs-only comparison for T values brings a benefit to almost every query from the RFID workload. Figures 4.24, 4.25, and 4.26 show the performance when additionally the query optimizations for pushing down selections and extended SIP are applied. Queries Q4, Q5, Q6, and Q9 are affected from the first optimization - pushing down selections, whereas the extended SIP optimization applies only to Q8. As explained in Section 4.4, if we take an index where the values are ordered by the *T* column, we can take advantage of the very fast sequential scan using start/stop conditions. This is the case for Q4, which performs after the optimization one order of magnitude better. For Q5, Q6, and Q9 this gain is not that significant (up to the factor of 3) because these queries have only one start or stop condition per index and thus cannot prune so big part of the indexes.

Applying extended SIP for Q8, i. e., we pass not only the reader values, but also the calculated timestamp values during the index scan (see Section 4.4) results in a speed up of a factor of 2 for this query.

Scalability

We observe that for the different pre-loaded database sizes shown in Figure 4.20, the query execution times of the mixed workload scale linearly. This is due to the bigger indexes that have to be loaded into main memory. Hence, only a smaller part of the indexes' working set is fitting into memory and the system performs more disk accesses. When doubling the initial data load size, all OLAP queries except for Q8 and Q11 grow by the factor of 2. Q8 and Q11 require scans of large amounts of intermediate data (which won't fit entirely into memory at some point) and also grow linearly, but with a higher slope, i. e., by a factor of 3 and 4. The OLTP queries are executed very often in the course of the benchmark. The interaction of growing execution times for the concurrent OLAP queries and the increasing lack of memory affects their response times greater than those of the OLAP queries. Further, there is a difference between the average values and the 95th percentile values. We determined that the 80th percentile is very close to the average value, i. e., only 20% of the queries have a higher response time. This is due to the fact that we query recent events with a probability of 80%; thus 20% of the queries are likely to perform disk access.

The experimental analysis shows that the dedicated Triple Store outperforms the examined general-purpose DBMSs regarding the event processing throughput as well as the query performance. Further, scaling the number of stored events, our approach can still provide the desired insert throughput.

4.6 Related Work

In the context of RDF, a variety of architectures which are optimized to naturally handle the triple-structured datasets have been proposed [32, 40, 55]. Those datasets are, in this respect, somewhat similar to RFID event data. The early open-source system Jena [32] uses property-clustered tables (triples grouped by the same predicate) for the storage of RDF data. In [55], a storage scheme called hexastore is presented, which allows for fast and scalable query processing. The RDF data is indexed in six possible ways, one for each possible ordering of the three RDF elements. The RDF-3X engine by Neumann et al. [40] – on which the RFID Triple Store is based – is a dedicated system for efficiently storing and querying RDF data. In RDF-3X, the triples are stored in one huge table which is exhaustively indexed. Furthermore, indexes are heavily compressed. RDF-3X achieves a very good query performance for read-mostly workloads in RDF scenarios by optimizing join orders so as to enable very fast merge join processing. Several interesting similarities between RDF and RFID data inspired us to use the approach of RDF-3X as a base for the RFID Triple Store.

4.7 Summary and Conclusions

In this chapter we presented a new dedicated solution for storage of RFID data, the RFID Triple Store. We showed that the system can easily handle the high insertion rates that are typical for object tracking applications. As was noted in Chapter 2, a database system should be able to cope with ca. 2000 events arriving per second for large enterprises. Our measurements showed that the RFID Triple Store can sustainably cope with an event frequency of 2500 events per second. We addressed the characteristics of RFID data in the architectural design of the RFID Triple Store. The RFID-aware indexes allow for efficient event processing. The performance for data extraction meets as well the requirements of an RFID scenario. The query engine greatly benefits from 15 clustered indexes providing all possible orderings of the triples table and from the order-preserving data dictionary, which speeds up the execution of range queries. Further optimizations of the query engine in order to exploit the RFID specific features brought an additional query acceleration. Our experimental study shows that our system can achieve a significantly higher throughput for event processing and a better query performance compared to general-purpose DBMSs.

Overall, the the RFID Triple Store approach clearly has the potential to meet the requirements of today's large-sized enterprises.

5 Distributed RFID Data Management

In this chapter, we present mechanisms for distributed processing of RFID data. First, we show how to use the MapReduce paradigm to conduct distributed RFID management. We evaluate and analyze the MapReduce performance. Second, an approach for distributed query processing on RFID data is presented, which uses the Triple Store discussed in Chapter 4.

5.1 Using MapReduce for the Management of RFID Data

Distributed processing paradigms like MapReduce [16] are gaining more and more attention in different application fields where large data sets are being processed. These include warehousing [53], extract-transform-load (ETL) tasks [17], graph [51] and software mining solutions [47]. Further, especially in the e-science community, MapReduce is increasingly being used for processing massive amounts of data.

MapReduce is an algorithmic concept which builds the basis for a huge variety of algorithms. It consists of two main steps: **map** and **reduce**. The MapReduce framework takes care of the automatical parallelization of the map and reduce functions, the partitioning of the input data, the scheduling of the program's execution and the managing of the data flow across the nodes in the cluster. The implementation of the map and reduce procedures is provided by the user.

In the map step, the input records are filtered, pruned or if possible divided into smaller sub-problems. The original input records are assigned to the map tasks by the MapReduce scheduler. A task in this context is every parallel executed instance created by the invocation of the map or reduce function respectively. The input file(s) are partitioned in equi-sized blocks of a pre-defined size. The generated map output is a set of intermediate "tuples" in the form of newly calculated key/value

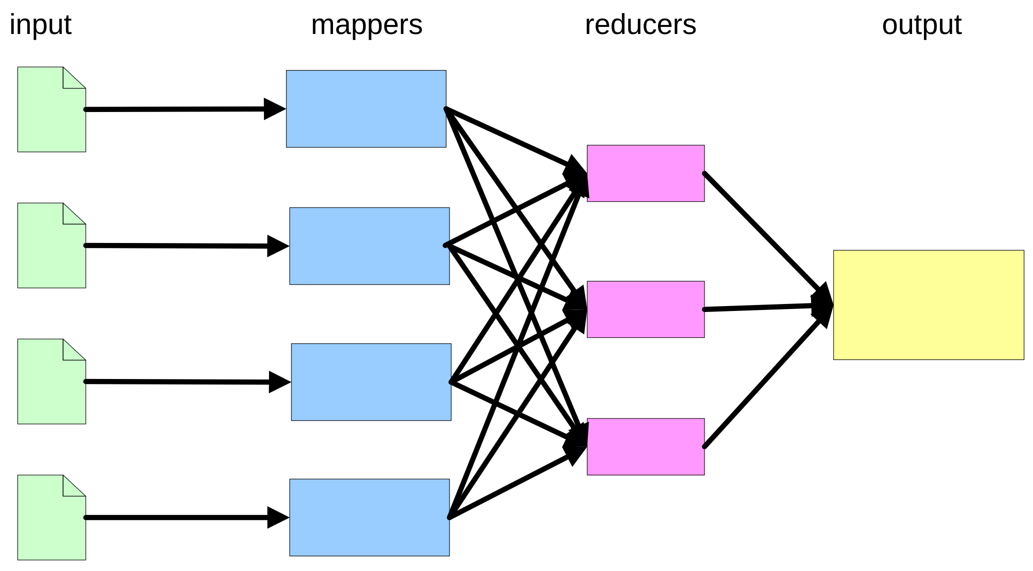


Figure 5.1: The MapReduce paradigm.

pairs. They are then redistributed (usually using hash partitioning) among the reducers. A typical hash function is, for instance: *hash(key)* **mod** *R*, where *key* is the hashed item key and *R* is the number of reducers. All map output tuples with the same hash value are processed by the same reducer. Therefore, the map function finishes with the creation of a number of output files, one for each reducer. Before the reduce phase starts, the relevant intermediate files (generated by the map functions) are transferred over the network from the maps' nodes to the reducers' nodes. Thus, the smaller the output from the map function, the more efficient this transfer step is due to the limitation of the network bandwidth.

In the reduce step, the map output is further processed and combined until the original task is solved. The reducers' output is written to files, one per reducer.

Sometimes more than one map and reduce phases may be needed for the completion of a single MapReduce program. Thus, the output of the reducers can be passed to another map task. Distributed processing of the map and the reduce steps is highly exploited. A sketch of the MapReduce paradigm can be seen in Figure 5.1. As illustrated, the map function retrieves its portion of the input, processes it and generates the output for the reducers, which produce the final output. Note that each of the m map tasks produce r output files, each assigned to a different reducer. This results in a total of $m * r$ produced files after the map phase.

For the management of RFID data, MapReduce [16] provides a feasible approach as it incurs practically no overhead for loading and storing the data. While the data staging process causes a major overhead in relational DBMSs (because of heavy index updates), using MapReduce we can simply store the log files containing the sensor events. These could be the direct output produced by the RFID sensors. Standard open-source MapReduce implementations do not provide the ability to index the data. However, there exist MapReduce extensions [18, 19] using built-in indexes that are implemented as user-defined functions. Furthermore, for analytical queries (as used in RFID applications), MapReduce proves a good performance [16]. To get an idea of the performance of MapReduce for managing RFID data, we designed our query workload using Hadoop [10] – an open-source implementation of MapReduce.

We found that there are some accruing fixed costs for the Hadoop framework due to synchronization overhead comprised of monitoring and starting the tasks on all nodes. Further, for each MapReduce program, in the map phase the whole (possibly huge) input has to be scanned, which makes the framework less appropriate for interactive workloads. DeWitt et al. confirm this observation in [41] by analyzing and comparing the performance of MapReduce (its Hadoop implementation) and parallel DBMSs. The results of the benchmark showed that both parallel database systems outperformed Hadoop when regarding different query tasks. For the data loading experiment, however, the opposite was shown: the MapReduce approach outperforms both distributed DBMSs by up to an order of magnitude. This is due to the fact that each input file is simply copied from the local disk into the Hadoop distributed file system, as opposed to a complex data staging procedure. Since we need fast event processing and since the MapReduce paradigm is known to perform good for analytical workloads we consider managing RFID data using Hadoop.

In this chapter, we analyze the performance of Hadoop regarding an RFID scenario. We implement a typical RFID query workload using Hadoop and measure the query response time.

5.1.1 A Short Introduction to Hadoop – An Implementation of MapReduce

Hadoop [10], developed by the Apache Software Foundation, is the most popular open-source implementation of the MapReduce framework. It is written in Java and thus platform independent. Hadoop comes with an own file system: the Hadoop Distributed File System (HDFS), which was derived by the Google File System (GFS) [26].

HDFS

HDFS is the distributed storage used by Hadoop. The map input and the reduce output are stored in the HDFS and they are thus accessible from every node in the cluster. HDFS is designed to be fault-tolerant and compensates hardware failures, which are very probable in huge clusters. This is achieved by replicating data blocks according to a specified replication factor. Files are split into data blocks by the HDFS. In contrast to common file systems, the size of the data blocks in HDFS is considerably higher: 64 MB. The reason for this is to decrease the amount of references (to the single blocks) that have to be stored and managed by a central instance. A HDFS cluster consists of a **NameNode** that manages the client access to files (the references to data blocks) as well as the file system's namespace and a number of **DataNodes**, one per cluster node, where the node's local data is stored. More details on the architecture of HDFS can be found in [10].

Processing a Hadoop job

A Hadoop job can be processed by committing it to a central instance called **Job-Tracker**. It supplies components for tracking the progress of tasks and getting the cluster's status information. The input records cannot be handled directly by the map procedure, since it expects key/value pairs. Therefore, the input is first split up according to an input specification described by the **InputFormat** class. This is responsible for parsing the input, splitting it up into logical instances (e. g., tuples) and providing a function for reading the records. A commonly used split function partitions the file into lines and uses the line number as a key and the content of the line as a value in the key/value pair. It is a primary goal of the HDFS to avoid transferring data between the nodes and it provides mechanisms for the applications to rather bring the processing task (the map tasks) to the data. The JobTracker tries to distribute the map tasks on the DataNodes such that no data is shuffled. The generated key/value pairs are assigned to the according mapper, where they are further processed and new key/value pairs are produced as output. Note that the input and output key/value pairs can have different data types, but all input and all output pairs have the same data type. As mentioned before, the default partitioning of the input uses the line number as a key and the line content as a value. This behavior can be, however, customized for each use case. For instance, a convenient way of representing our event triples is the following: the key value consists of an EPC data type and the value part has a complex data type consisting of the reader and timestamp entries. We apply this input format when implementing our queries.

The map output is then partitioned by the **Partitioner** instance. The default partition function is a hash function applied to the key. The user can define a custom partitioning function as well. All tuples with the same key are assigned to the same reducer. After transferring all data belonging to the same reducer to the reducer's local disk, the records are sorted by key by the Hadoop framework. The key/value pairs passed to the reducer have the form $\langle key, \{v : value\ v\ belongs\ to\ key\}\rangle$. Similar to the mapper, the output of the reducer task can have a different data type than that of its input. For instance, if the map input contains the following input pairs $\langle EPC, RdrTS\rangle$, the corresponding output format can contain only reader IDs as a key: $\langle Rdr\rangle$. The reducer's output is written to files residing in the HDFS and is formatted according to the **OutputFormat** class. It specifies that each key/value pair is written in one line. The number of output files created is equal to the number of reducers. The records in each output file are sorted by the key value.

In order to minimize the data transfer between the nodes, combiner tasks can be inserted between the mapper and the reducer tasks. If a combiner is used, the key/value pairs of the mapper are not written immediately to disk, but are first collected in memory in form of lists – one per key. These lists are passed to the combiner's reduce method, which can be seen as a local reducer. It can aggregate the output of the mapper before it is distributed to the reducer nodes. By doing this, the key/value output pairs are reduced and only this smaller output is written to disk. Therefore, using an appropriate combiner accelerates Hadoop's performance. More information can be found in [10].

5.1.2 Implementing RFID Queries Using Hadoop

In this section, we show how our RFID queries are implemented using Hadoop. A primary goal is to reduce the amount of data that is redistributed over the network. If this is not considered in the query implementation, Hadoop's performance will decrease and the overhead of distributing the data could outweigh the advantages of parallel processing. We therefore try to keep the size of the map step results as small as possible. The map, combine, and reduce functions for each query are described in the following. This section builds upon the work in [33] which was supervised by the author of this thesis.

An example data set

In order to illustrate the input and output of the map and reduce procedures for each query, the data set in Table 5.1 is used as an example. It depicts four objects (with the EPC codes 1, 2, 3, and 4) that pass through different scanners at different points in time. The time intervals in which the EPCs are scanned are chosen to partly overlap for a better demonstration of the RFID range queries – in order to have more matches. The data is ordered by EPC.

Q1: Last location of an object

Mapper: The mapper for Q1 is shown in Algorithm 10. It first selects those events which contain the requested object ID (*myEpc*). All other triples are omitted and not processed further.

Algorithm 10: Mapper Q1

```
input  : ⟨EPC key, RdrTS value⟩, EPC myEpc
output: ⟨EPC key, RdrTS value⟩
1 if key = myEpc then
2     write(⟨key, value⟩);
3 end
```

EVENTS		
epc	**rdr**	**ts**
1	1	1
1	2	2
1	3	5
1	2	6
1	1	10
2	3	2
2	4	4
3	1	2
4	4	4
4	5	6
4	1	15

Table 5.1: An example data set.

If the given EPC is 1 for our example in Figure 5.1, the mapper output is: $\langle 1, [1,1] \rangle$, $\langle 1, [2,2] \rangle$, $\langle 1, [3,5] \rangle$, $\langle 1, [2,6] \rangle$, $\langle 1, [1,10] \rangle$.

Combiner: A local maximum of the timestamp value can be calculated by the combiner as shown in Algorithm 11. This saves some timestamp value comparisons for the reducer. However, since the average object's path in our experimental scenario has the length of 19 (as described in Chapter 4), the combiner's work does not have much impact on the performance. There are as many combiner outputs as reducers in the cluster.

Algorithm 11: Combiner Q1

```
input : ⟨EPC key, List⟨RdrTS⟩ values⟩
output: ⟨EPC key, List⟨RdrTS⟩ values⟩
1 maxTs, maxRdr = −∞;
2 for v: values do
3     if v.ts > maxTs then
4         maxRdr = v.rdr;
5         maxTs = v.ts;
6     end
7 end
8 write(⟨key, pair(maxRdr, maxTs)⟩);
```

Reducer: Since all selected event tuples have the same key (the selected EPC), they will end up in the same reducer. The reducer for Q1, which is depicted in

Algorithm 12, needs therefore to compare all timestamps, to choose the latest one, and to return the sensor at which the object was scanned last.

Algorithm 12: Reducer Q1

```
input : $\langle$EPC $key, List\langle$RdrTS$\rangle$ $values\rangle$
output: $\langle$Rdr $value\rangle$
1 $maxTs, maxRdr = -\infty$;
2 for $v$: $values$ do
3     if $v.ts > maxTs$ then
4         $maxRdr = v.rdr$;
5         $maxTs = v.ts$;
6     end
7 end
8 write($\langle maxRdr\rangle$);
```

For our example, the reader with ID 1 is the output.

Q2: The pedigree (complete path) of an object

Mapper: The mapper of Q2 shown in Algorithm 13 selects those events which contain the requested object ID (*myEpc*). All other triples are omitted and not processed further. In order to have the keys sorted by the timestamp value (that means it should be partitioned by timestamp), we use the timestamp as a key and the reader as a value in the output key/value pair. This yields an appropriate order of the timestamps for the reduce step. If $myEpc = 1$ the output of the mapper for our example is: $\langle 1,1\rangle$, $\langle 2,2\rangle$, $\langle 5,3\rangle$, $\langle 6,2\rangle$, $\langle 10,1\rangle$.

Algorithm 13: Mapper Q2

```
input : $\langle$EPC $key$, RdrTS $value\rangle$, EPC $myEpc$
output: $\langle$TS $key$, Rdr $value\rangle$
1 if $key = myEpc$ then
2     write($\langle value.ts, value.rdr\rangle$);
3 end
```

Reducer: The mapper streams the result tuples ordered ascendingly by their timestamp values. The reducer for Q2 depicted in Algorithm 14 just passes the sensor IDs (identity reducer). Therefore the output of the reducer is: $\langle 1\rangle$, $\langle 2\rangle$, $\langle 3\rangle$, $\langle 2\rangle$, $\langle 1\rangle$. Since the reducer's output is sorted by timestamp but partitioned in different output files, we need to execute one final merge step at the end to globally merge the different outputs. Hadoop provides an appropriate function call for this task.

Algorithm 14: Reducer Q2

```
input : $\langle$TS $key$, $List\langle$Rdr$\rangle$ $values\rangle$
output: $\langle$Rdr $value\rangle$
for $v$: $values$ do
    write($\langle v\rangle$);
end
```

Q3: The number of objects scanned by a certain sensor

Mapper: The mapper for Q3 which is shown in Algorithm 15 selects all the events that were scanned by the given reader *myRdr*. We use the sensor ID as the key and the EPC as the value in the key/value output of the mapper. This will guarantee that all values with the same sensor ID are processed by the same reducer. If we choose for the given example that the ID of *myRdr* is 2, then the output of the map function is: $\langle 2, 1\rangle$, $\langle 2, 1\rangle$.

Algorithm 15: Mapper Q3

```
input : $\langle$EPC $key$, RdrTS $value\rangle$, Rdr $myRdr$
output: $\langle$Rdr $key$, EPC $value\rangle$
if $value.rdr = myRdr$ then
    write($\langle value.rdr, key\rangle$);
end
```

Algorithm 16: Reducer Q3

```
input : $\langle$Rdr $key$, $List\langle$EPC$\rangle$ $values\rangle$
output: $\langle$Rdr $key$, count $values\rangle$
$Set\ s = \{\}$;
for $v$: $values$ do
    $s = s \cup \{v\}$;
end
write($\langle key, |s|\rangle$);
```

Reducer: The reducer for Q3 performs duplicate elimination and counts all the different EPCs for the particular reader. Our implementation uses a set to eliminate duplicates as shown in Algorithm 16. For the chosen example, the reducer returns only one group: $\langle 2, 1\rangle$.

Q4: A list of objects scanned by a sensor within a time interval

Mapper: The mapper for Q4 described in Algorithm 17 selects all events read by the given sensor *myRdr* and checks if the events were created within the given time interval $]t_0, t_1[$. Given the reader ID $myRdr = 1$ and the time interval $]1, 15[$ the result of the map function is: $\langle 1 \rangle$, $\langle 3 \rangle$ generated from the tuples $\langle 1, 1, 10 \rangle$, $\langle 3, 1, 2 \rangle$.

Algorithm 17: Mapper Q4

```
input : $\langle$EPC $key$, RdrTS $value\rangle$, Rdr $myRdr$, TS $t_0$, TS $t_1$
output: $\langle$EPC $key\rangle$
1 if $value.rdr = myRdr \wedge value.ts > t_0 \wedge value.ts < t_1$ then
2     write($\langle key \rangle$);
3 end
```

Reducer: The mapper does all the work for this query so only correct tuples, i. e., tuples that fulfill the select conditions, arrive at the reducer. Since the query performs duplicate elimination on the key (EPC) values, each reducer just returns one key value per key group. Therefore, the reducer for Q4 is an identity-reducer that outputs only one key for each key/value group and performs this way duplicate elimination. The output for the example query is thus: $\langle 1 \rangle$, $\langle 3 \rangle$.

Q5: A list of objects which were scanned by sensor rdr_1 after a time threshold and by sensor rdr_2 before a time threshold

Mapper: The mapper for Q5 depicted in Algorithm 18 selects all events that were read by one of the two sensors rdr_1 or rdr_2, such that the timestamps of the objects scanned by rdr_1 are greater than t_0 and the timestamps of the objects scanned by rdr_2 are smaller than t_1. This query filters objects that were scanned by both readers within the given time constraints without regarding the sequence in which these events passed the readers. Consider, that the sensors have a specific semantic, like entry and exit. This query determines for example the objects that passed an entry sensor after 8 o'clock in the morning and an exit sensor before 12 o'clock in the morning, i. e., all objects produced before noon. We can calculate the join condition (determining that the objects were scanned by both sensors) only in the reducer since not all tuples with the same EPC will end up in the same mapper. Thus, for every object, the reducer has to validate whether the object was scanned by both sensors. Therefore, we take the EPC as the key. For the example data in Figure 5.1 given the sensor IDs 1 and 2 and the timestamps 1 and 15 for the variables t_0 and t_1 respectively, the mapper output is: $\langle 1, [2, 2] \rangle$, $\langle 1, [2, 6] \rangle$, $\langle 1, [1, 10] \rangle$, $\langle 3, [1, 2] \rangle$.

Reducer: The reducer for Q5 checks for the presence of every object at both sensors (join condition). In this case the object's ID, the EPC, is written to the output.

Algorithm 18: Mapper Q5

```
input  : $\langle$EPC $key$, RdrTS $value\rangle$, Rdr $rdr_1$, Rdr $rdr_2$, TS $t_0$, TS $t_1$
output: $\langle$EPC $key$, RdrTS $value\rangle$
if ($value.rdr = rdr_1 \wedge value.ts > t_0$) $\vee$ ($value.rdr = rdr_2 \wedge value.ts < t_1$) then
    write($\langle key, value\rangle$);
end
```

Algorithm 19 shows how this is done in Hadoop. The output for our example is the EPC with ID $\langle 1\rangle$.

Algorithm 19: Reducer Q5

```
input  : $\langle$EPC $key$, $List\langle$RdrTs$\rangle$ $values\rangle$, Rdr $rdr_1$, Rdr $rdr_2$
output: $\langle$EPC $key\rangle$
$isrdr_1, isrdr_2 = false$;
for $v$: $values$ do
    if $v.rdr = rdr_1$ then
        $isrdr_1 = true$;
    end
    if $v.rdr = rdr_2$ then
        $isrdr_2 = true$;
    end
end
if $isrdr_1 \wedge isrdr_2$ then
    write($\langle key\rangle$);
end
```

Q6: A list of objects which were scanned first by sensor rdr_1 and then by sensor rdr_2 within a time interval

Mapper: The mapper for Q6 selects all events that were read by one of the two readers rdr_1 and rdr_2 in the given interval $]t_0, t_1[$. The difference to Q5 is that Q6 considers the order in which the events passed the readers, e. g., first rdr_1, then rdr_2. The mapper's implementation is identical to Algorithm 18. Given the example parameters $rdr_1 = 2$, $rdr_2 = 3$, $t_0 = 1$, and $t_1 = 10$, the map function returns: $\langle 1, [2, 2]\rangle$, $\langle 1, [3, 5]\rangle$, $\langle 1, [2, 6]\rangle$, $\langle 2, [3, 2]\rangle$.

Combiner: The combiner for Q6 is depicted in Algorithm 20 and selects the minimum timestamp for the first reader and the maximum timestamp for the second reader. This way it saves some timestamp comparisons that will be done by the reducer otherwise.

Algorithm 20: Combiner Q6

```
input : ⟨EPC key, List⟨RdrTS⟩ values⟩, Rdr $rdr_1$, Rdr $rdr_2$
output: ⟨EPC key, List⟨RdrTS⟩ values⟩
$currts_1 = \infty$;
$currts_2 = -\infty$;
$isrdr_1, isrdr_2 = false$;
for v: values do
    if $v.rdr = rdr_1$ then
        $isrdr_1 = true$;
        if $v.ts < currts_1$ then
            $currts_1 = v.ts$;
        end
    end
    if $v.rdr = rdr_2$ then
        $isrdr_2 = true$;
        if $v.ts > currts_2$ then
            $currts_2 = v.ts$;
        end
    end
end
if $isrdr_1$ then
    write(⟨key, pair⟨$rdr_1$, $currts_1$⟩⟩);
end
if $isrdr_2$ then
    write(⟨key, pair⟨$rdr_2$, $currts_2$⟩⟩);
end
```

Reducer: The reducer for Q6 (Algorithm 21) verifies again that the object was scanned by both sensors and that the timestamps are in the correct order. For our example the reducer returns the EPC with ID ⟨1⟩.

Q7: The number of objects which were scanned first by sensor rdr_1 and then by sensor rdr_2 within a time interval

This query requires two MapReduce runs since the reducer has to operate on two different key/value pairs in order to count all object IDs. The first run is identical to query Q6. The second one processes the output of the first run and only performs the counting of the tuples.

Mapper of the first run: The first mapper for Q7 selects all events that were read by one of the two readers rdr_1 and rdr_2 in the given interval $]t_0, t_1[$ (code identical to Algorithm 18).

Algorithm 21: Reducer Q6

```
input : $\langle$EPC $key$, $List\langle$RdrTS$\rangle$ $values\rangle$, Rdr $rdr_1$, Rdr $rdr_2$
output: $\langle$EPC $key\rangle$
$currts_1 = \infty$;
$currts_2 = -\infty$;
$isrdr_1, isrdr_2 = false$;
for $v$: $values$ do
    if $v.rdr = rdr_1$ then
        $isrdr_1 = true$;
        if $v.ts < currts_1$ then
            $currts_1 = v.ts$;
        end
    end
    if $v.rdr = rdr_2$ then
        $isrdr_2 = true$;
        if $v.ts > currts_2$ then
            $currts_2 = v.ts$;
        end
    end
end
if $isrdr_1 \wedge isrdr_2 \wedge (currts_1 < currts_2)$ then
    write($\langle key \rangle$);
end
```

Reducer of the first run: The first reducer of Q7 verifies that the object was scanned by both readers rdr_1 and rdr_2 in the correct order (code identical to Algorithm 21). The output of the reducer for the example given in Q6 is: $\langle 1 \rangle$.

Mapper of the second run: The second mapper for Q7 in Algorithm 22 has to count all records. It therefore creates a dummy key value for all tuples and passes on a value of 1 for efficient counting. For our example, this produces the output: $\langle 0, 1 \rangle$.

Algorithm 22: Mapper Q7 / Second run

```
input : $\langle$EPC $key\rangle$
output: $\langle$dummy $key$, count $value\rangle$
write($\langle 0,1 \rangle$);
```

Reducer of the second run: The second reducer for Q7 sums up the values globally and provides the sum as output (Algorithm 23). In our case the output is: $\langle 1 \rangle$, as only one object fulfills the query. We can also use a combiner for this

MapReduce job that sums up the values locally, similar to that for Q3.

Algorithm 23: Reducer Q7 / Second run

```
input : ⟨dummy key, List⟨count⟩ values⟩
output: ⟨count value⟩
1 counter = 0;
2 for v: values do
3     counter+ = v;
4 end
5 write(⟨counter⟩);
```

Q8: A list of objects that were at sensor *rdr*, together with an object *myEpc* (suspected to be contaminated) within a certain time interval [*myEpc.ts* − *t*, *myEpc.ts* + *t*] where *t* is the given interval limit (contamination query)

This query is a heavy OLAP query. A naïve approach of implementing it using Hadoop will end up in using more than one mapper and reducer runs. This is expensive for Hadoop since the intermediate results of the reduce phases are written to disk and have to be read from disk again for the next MapReduce job. For this reason, we modify this approach slightly and design a solution that needs only one map and reduce cycle. The idea is to bring all events produced by one sensor to one reducer and have them sorted by their timestamps, so that we iterate only once through the list and stop when the timestamp is out of range. Therefore, it is first determined whether the contaminated object was scanned by the particular sensor and if this holds, all other objects within the given time range are selected. We take the following fact into account: we search for all events in the time range of the contaminated item, i. e., the interval from *myEpc.ts* − *t* to *myEpc.ts* + *t* where *myEpc.ts* is the timestamp of the contaminated object and *t* is the given time range. After sorting the results by timestamp we iterate through the list and select all possibly contaminated objects (those which lie within the time interval).

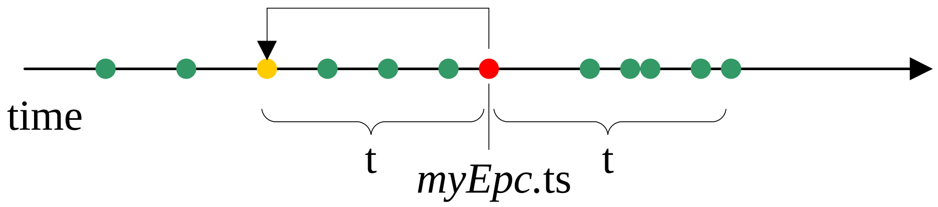


Figure 5.2: Shifting of the time interval of the contaminated object *myEpc* (red point to yellow point).

Mapper: The mapper for Q8 given in Algorithm 24 selects the given contaminated object with EPC value *myEpc* and sets its timestamp to the beginning of the time interval: $myEpc.ts - t$, as illustrated in Figure 5.2. In order that the reducer gets the data partitioned by sensor, we choose as a key the $pair\langle \text{Rdr } key, \text{TS } value\rangle$. We rewrite the Partitioner class to consider only the first part of the key for partitioning, i. e., the reader ID and not the timestamp. Each reducer gets all the values for a particular reader. Since the framework sorts the input for the reducers by key, the entries are sorted by timestamp (since the reader part is always the same). The value part is the $pair\langle \text{EPC } key, \text{TS } value\rangle$, which contains the same timestamp as the key, and is therefore also sorted by timestamp. To give an example for this query, we assign the ID 1 to the given contaminated EPC *myEpc* and consider the given interval limit to be 1. Note that the interval is then constructed to be symmetric around the given interval limit *t*: the time interval is therefore [0, 2]. The output of the map task as key/value pairs for the test data in Table 5.1 and the given parameters is:

k	[1,0]	[2,1]	[3,4]	[2,5]	[1,9]	[3,2]	[4,4]	[1,2]	[4,4]	[5,6]	[1,15]
v	[1,0]	[1,1]	[1,4]	[1,5]	[1,9]	[2,2]	[2,4]	[3,2]	[4,4]	[4,6]	[4,15]

Algorithm 24: Mapper Q8

```
input : ⟨EPC key, RdrTS value⟩, EPC myEpc, Interval t
output: ⟨pair⟨Rdr key, TS value⟩, pair⟨EPC key, TS value⟩⟩
1 if $key = myEpc$ then
2     $value.ts = value.ts - t$;
3 end
4 write(⟨pair⟨value.rdr, value.ts⟩, pair⟨key, value.ts⟩⟩);
```

Reducers' Input

reducer no.	key	values
1	⟨1,0⟩	⟨1,0⟩, ⟨3,2⟩, ⟨1,9⟩, ⟨4,15⟩
2	⟨2,1⟩	⟨1,1⟩, ⟨1,5⟩
3	⟨3,4⟩	⟨2,2⟩, ⟨1,4⟩
4	⟨4,4⟩	⟨4,4⟩, ⟨2,4⟩
5	⟨5,6⟩	⟨4,6⟩

Table 5.2: The output of the map function after partitioning, grouping, and sorting which is also the input for the reducer function. The value tuples are sorted by timestamp.

Reducer: After partitioning, grouping and sorting, each reducer gets the output of the map function as depicted in Table 5.2. The key pairs contain the reader and its corresponding timestamp value: $pair\langle value.rdr, value.ts\rangle$. The value pairs *pair*⟨*key*, *value.ts*⟩ that are grouped by the reader are sorted by timestamp. Every reducer processes all events scanned by one particular sensor as described in Algorithm 25. In line 2, the contamination interval *continterval* is set to an invalid value. The reducer first checks if the contaminated item is amongst the events read by the sensor (line 4 of the algorithm). If this holds, the contamination interval is updated to the timestamp of the contaminated item. Now, all items within the time interval of the contaminated object (calculated by the condition in line 6) are added to the output. All other objects are omitted. For this procedure, the reducer function has to scan the complete list of events once for each reader. After the reducers complete their execution the result of the example query is the triple $\langle 3, 1, 2\rangle$ and the output is therefore $\langle 3\rangle$.

Algorithm 25: Reducer Q8

```
input : ⟨pair⟨Rdr key, TS value⟩, List⟨EPCTS⟩ values⟩, EPC myEpc, Interval t
output: ⟨EPC key⟩
1 $wholerange = 2 \cdot t$;
2 $continterval = -wholerange - 1$;
3 for v: values do
4     if $v.epc = myEpc$ then
5         $continterval = v.ts$;
6     else if $v.ts \leq continterval + wholerange$ then
7         write(⟨v.epc⟩);
8     end
9 end
```

Q9: Listing the number of objects per reader and timestamp, which passed in a certain time interval (e.g., to identify production peaks)

Mapper: The mapper in Algorithm 26 selects all events that were generated in the given time interval $]t_0, t_1[$. The output is split by the sensor ID over the different reducers. If we take as interval]1,4[from our example data set we get the following output from the mapper: $\langle[2,2],1\rangle$, $\langle[3,2],1\rangle$, $\langle[1,2],1\rangle$.

Algorithm 26: Mapper Q9

```
input : ⟨EPC key, RdrTS value⟩, TS $t_1$, TS $t_2$
output: ⟨pair⟨Rdr key, TS value⟩, count counter⟩
1 if $value.ts > t_1 \wedge value.ts < t_2$ then
2     write(⟨pair⟨value.rdr, value.ts⟩, 1⟩);
3 end
```

Combiner: The combiner counts the tuples that were produced by the mappers locally, i. e., per reader and timestamp group from each mapper. The code is analogous to Algorithm 23 with the difference that instead of a list of counters, the function takes $\langle pair\langle rdr, ts\rangle\rangle$ as an input. Further, the pair $\langle pair\langle rdr, ts\rangle\rangle$ and the respective counter value per pair is returned.

Reducer: The reducer counts the tuples originating from the combiner globally and outputs the count result grouped by reader and timestamp. The code is analogous to Algorithm 23 with the difference that instead of a list of counters, the function takes $\langle pair\langle rdr, ts\rangle\rangle$ as an input. Further, the pair $\langle pair\langle rdr, ts\rangle\rangle$ and the respective counter value per pair is returned. The output for our example is: $\langle[2,2],1\rangle$, $\langle[3,2],1\rangle$, $\langle[1,2],1\rangle$.

Q10: Listing the number of all objects scanned at any of a given set of readers within a time interval, grouped and ordered by reader, and a time interval of a second

Mapper: The mapper for Q10 is given in Algorithm 27. It selects all events that were produced by one of the given sensors $\{rdr \mid rdr \in \text{Rdr}\}$ in the pre-defined time interval $]t_0, t_1[$. It passes key/value pairs consisting of the reader and timestamp as a key and 1 as a value that is to be counted by the reducer. The same object is not supposed to be scanned at the same reader at the exact same time again, since we assume that the data is cleaned beforehand. Therefore, we do not need to perform duplicate elimination for this query. For our example, if we take all five existing sensor IDs as input and the time interval]1,6[we get the following mapper output: $\langle[2,2],1\rangle$, $\langle[3,5],1\rangle$, $\langle[3,2],1\rangle$, $\langle[4,4],1\rangle$, $\langle[1,2],1\rangle$, $\langle[4,4],1\rangle$.

Algorithm 27: Mapper Q10

```
input : ⟨EPC key, RdrTS value⟩, {rdr | rdr ∈ Rdr}, TS $t_0$, TS $t_1$
output: ⟨RdrTS value, counter⟩
1 if $value.rdr \in \{rdr \mid rdr \in Rdr\} \wedge value.ts > t_0 \wedge value.ts < t_1$ then
2     write(⟨value, 1⟩);
3 end
```

Combiner: The combiner for Q10 counts locally the number of events per reader. The code is analogous to Algorithm 23 with the difference that instead of a list of counters, the function takes $\langle pair\langle rdr, ts\rangle\rangle$ as an input. Further, the pair $\langle pair\langle rdr, ts\rangle\rangle$ and the respective counter value per pair is returned.

Reducer: The reducer for Q10 counts the number of events per reader and timestamp globally. The code is analogous to Algorithm 23 with the difference that instead of a list of counters, the function takes $\langle pair\langle rdr, ts\rangle\rangle$ as an input. Further, the pair $\langle pair\langle rdr, ts\rangle\rangle$ and the respective counter value per pair is returned. For our example the reducers' output is: $\langle[2,2],1\rangle$, $\langle[3,5],1\rangle$, $\langle[3,2],1\rangle$, $\langle[4,4],2\rangle$, $\langle[1,2],1\rangle$.

Q11: Listing the number of all objects which were scanned within a time interval by the sensors rdr_1, rdr_2, and rdr_3 in this order, aggregated per second

The processing of this query requires two different MapReduce jobs. The first run selects all triples that fulfill the conditions of the query. The reducer needs to work on all events for one object, i. e., the object ID is the key in the function. The second run is responsible for counting the result. The second reducer expects the timestamp value as a key since the occurrence of the timestamps has to be counted.

Mapper of the first run: This mapper depicted in Algorithm 28 selects all events that were scanned by the three sensors in the specified time range. Given the reader IDs 2, 3, and 1 and the time interval $]1,15[$ the mapper of the first run outputs: $\langle 1,[1,1]\rangle$, $\langle 1,[2,2]\rangle$, $\langle 1,[3,5]\rangle$, $\langle 1,[2,6]\rangle$, $\langle 1,[1,10]\rangle$, $\langle 2,[3,2]\rangle$, $\langle 3,[1,2]\rangle$.

Algorithm 28: Mapper Q11 / First run

```
input : ⟨EPC key, RdrTS value⟩, Rdr rdr1, Rdr rdr2, Rdr rdr3, TS t0, TS t1
output: ⟨EPC key, RdrTS value⟩
1 if $value.rdr = rdr_1 \land value.ts > t_0$
2 $\lor$ $value.rdr = rdr_2$
3 $\lor$ $value.rdr = rdr_3 \land value.ts < t_1$
4 then
5     write(⟨key, value⟩);
6 end
```

Reducer of the first run: The first reducer for Q11 given in Algorithm 29 has to determine the minimum timestamp ts_1 of events read by sensor rdr_1 and the maximum timestamp ts_3 of sensor rdr_3. For all events read by sensor rdr_2, we store the timestamp in order to check if it is within the interval $[ts_1, ts_3]$. If there is one event that fulfills the conditions then its timestamp and a count of 1 is written as output. For our example, if we apply the reducer algorithm for the triples with EPC $key = 1$: $\langle 1,[1,1]\rangle$, $\langle 1,[2,2]\rangle$, $\langle 1,[3,5]\rangle$, $\langle 1,[2,6]\rangle$, $\langle 1,[1,10]\rangle$ we see that the object with ID 1 was scanned by the reader IDs 2, 3, and 1 in the correct order in the given time interval. Therefore, the output of the reducer is the smallest timestamp for rdr_1 : $\langle 2,1\rangle$.

Mapper of the second run: The second mapper for Q11 is a simple identity mapper.

Reducer of the second run: The second reducer for Q11 counts the resulted timestamp groups. The code is analogous to Algorithm 23 except that the key is the timestamp value and the output contains the timestamp and count pairs. The output of the reducer from the second run is therefore: $\langle 2,1\rangle$.

Algorithm 29: Reducer Q11 / First run

```
input : ⟨EPC key, List⟨RdrTS⟩ values⟩, Rdr $rdr_1$, Rdr $rdr_2$, Rdr $rdr_3$
output: ⟨TS key, counter⟩
$ts_1 = \infty$ $ts_3 = -\infty$;
twoisin = false;
Set $ts_2 = \{\}$;
for v: values do
    if $v.rdr = rdr_1 \wedge ts_1 > v.ts$ then
        $ts_1 = v.ts$;
    else if $v.rdr = rdr_3 \wedge ts_3 < v.ts$ then
        $ts_3 = v.ts$;
    else
        $ts_2 = ts_2 \cup \{v.ts\}$;
    end
end
if $ts_1 < ts_3$ then
    for $t : ts_2$ do
        if $t \geq ts_1 \wedge t \leq ts_3$ then
            twoisin = true;
            break;
        end
    end
end
if twoisin then
    write(⟨$ts_1$,1⟩);
end
```

5.1.3 Performance Evaluation and Comparison

Hadoop setup

For our experimental study we use Hadoop version 1.0.1 running on Java 1.6.0. We installed Hadoop with the default configuration settings except for the following properties that were changed in order to get a better performance: (1) we use 512 MB data block size instead of the default 64 MB in order that the map instances process a bigger portion of the data at a time, (2) each Java task tracker child process can use up to 2560 MB heap size instead of the default 200 MB in order to hold as many intermediate results as possible in main memory. Further, we configured the system to run two map and two reduce instances in parallel on each node. For every Hadoop job one can additionally specify the overall number of map and reduce instances that should be launched. Note that these Hadoop parameters are just a hint for the framework and Hadoop decides how many map instances to run and how to allocate them on the nodes. According to the Hadoop documentation, one reducer per cluster node is a reasonable job configuration. We followed this advice and the number of reducers chosen by Hadoop matched exactly our input in the configuration file: the number of cluster nodes. The number of mappers is determined by the size of the input file(s). This size divided by the configured block size gives the number of mappers started by the framework for the particular job. All input and output files are stored in the HDFS. Two replicas per block were configured.

Node configuration

We use three different node configurations for the experiments: a 4-node, a 8-node, and a 16-node cluster. Each node is equipped with an Intel(R) Core(TM)2 Quad CPU and 7 GB RAM. HDFS is installed on a dedicated local 500 GB LVM volume. One distinguished cluster node is serving as a NameNode and a Jobtracker as well as a master and a slave. All other nodes are slaves (see the HDFS architecture in [10]).

Benchmark execution

For our experiments, we generate the amount of RFID data produced in one week in a world-wide company: 450 million events. This is the same data set we used for evaluating the performance of our Triple Store in Section 4.5. The event triple file loaded in HDFS has a total size of 50.8 GB, it contains 1000 different reader values, 22494720 different EPC values and 4500049 different timestamp values.

We implemented each query as a set of Hadoop jobs according to the descriptions above. We executed each query three times for each configuration (a 4-node, a 8-node, and a 16-node cluster) and report the average time of the runs. We further experiment with a smaller data size file containing only the events of the last production day (ca. 10 GB) in order to avoid that Hadoop scans the whole input for the time range queries. For the first setup, 100 mappers are running in the framework; for the second scenario, the mappers determined by Hadoop are only 19 due to the smaller input. The query results can be seen in Figures 5.3, 5.4 and 5.5.

Results and Discussion

As expected, the experimental results in Figure 5.3 show that Hadoop is not appropriate for OLTP processing. Q1 and Q2 select the last position of an object and the whole pedigree path of an object, respectively. Due to the fact that indexes cannot be used with plain Hadoop, the framework has to scan the whole input files in order to find a particular record, e. g., the particular EPC. The execution time of the queries is thus completely dominated by the map process of filtering the data and reading data from disk – that is, disk I/O. Therefore, OLTP queries, which are usually using fast indexing in common DBMSs, suffer badly in this context due to the lack of built-in indexes.

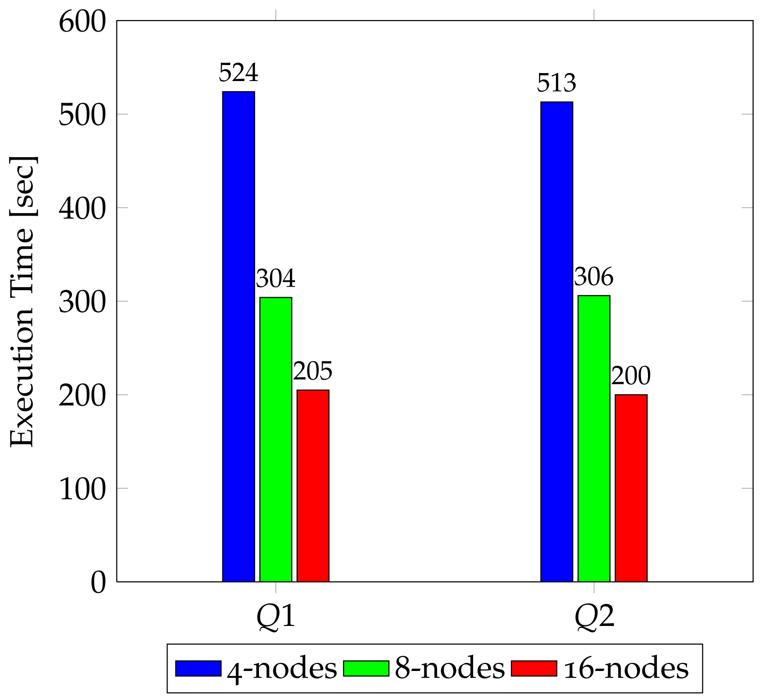


Figure 5.3: Query performance of the OLTP queries on different Hadoop clusters.

Further, we found that the fixed overhead of the MapReduce framework for starting all services, synchronizing the tasks and performing a disk access amounts ca. 25 seconds. This was also stated in [41]. For short-running queries, these warm-up costs can therefore dominate the execution time.

In Figure 5.4, the performance of the OLAP queries in our RFID benchmark for the one week data set are depicted. We notice that Q3, Q4, Q5, Q6, and Q9 show a similar query performance and investigate the effects on these queries' runtime first.

For Q3, the map function selects all events at one particular reader and the reduce function counts the outcoming events. The execution of the map function takes expressed as a percentage of the map phase longer than the reduce functions of the rest of the queries. This is due to the fact that this query uses only one reducer (the map output is partitioned on the sensor ID), which processes 1/1000 of the data (the events are equally distributed over 1000 readers).

Q4's functionality lies primarily in the mapper, whose execution time dominates completely the performance. It is interesting that even using the identity reducer an overhead of ca. 30 seconds is produced. This corresponds to the time for reading the output of the map tasks from disk and writing it to the reduce output, i. e., the I/O time. The query could be further optimized if we omit the reduce phase and use the map output as a final output of the query.

The actual join computation of Q5 is done by the reducer task, the map instances perform the selection conditions. Each map instance filters all objects that were scanned by one of the given readers in the desired time interval. The map phase has to process the whole input data, while the reduce phase takes only a fraction of the data, which again results in a longer map phase execution.

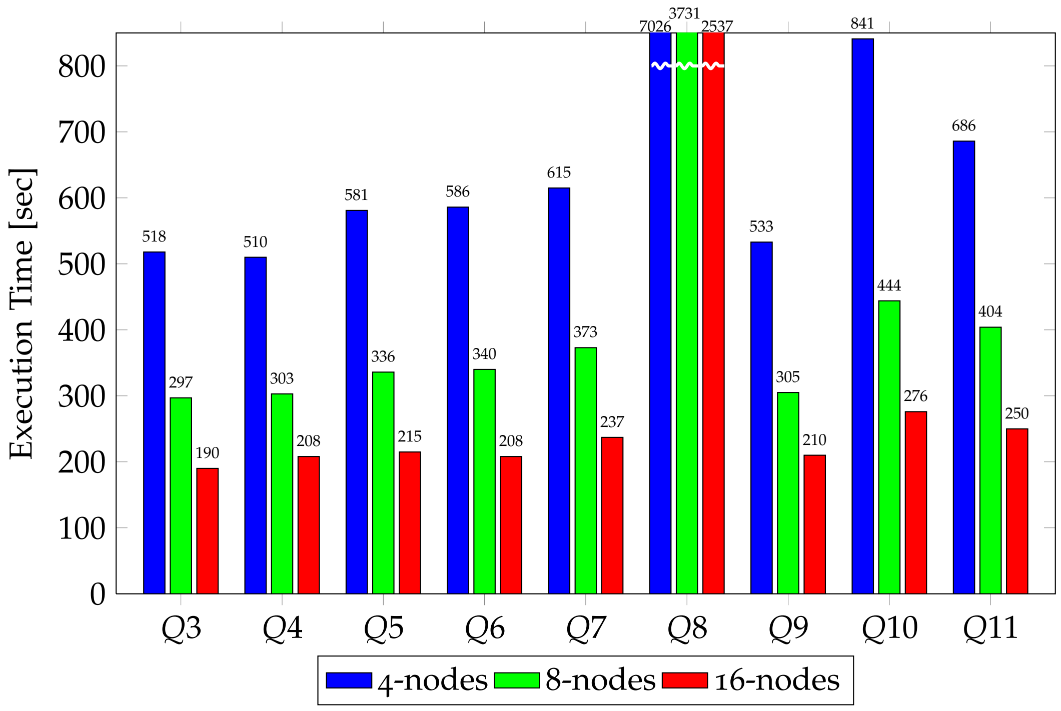


Figure 5.4: Query performance of the OLAP queries measured on the one week dataset, using different Hadoop cluster sizes.

For Q6, we extend the reduce procedure of Q5 by an additional condition, which guarantees that the items were first scanned by the first reader and then by the second reader. Consequently, this phase takes slightly longer than the reduce phase of Q5. Overall, that is the reason for the slight increase of the execution time of Q6 compared to Q5.

Q9 is first selecting all $\langle Rdr, TS \rangle$ pairs in the given time interval (done in the map phase) and is then grouping and counting the pairs in the reduce phase. Since the key is the sensor ID and in our test data set we have 1000 readers that produce events which are equally distributed over the time, the reducers get similar portions of the data.

Queries Q7, Q8, Q10, and Q11 take longer than the rest and we take a closer look at their map and reduce procedures.

As described in Section 5.1.2, Q7 needs two map and reduce phases. The first map and reduce run is identical to Q6; the second one does an additional counting of the output of Q6. Using more than one map and reduce phases, increases the execution time because of the intermediate disk writes. The overhead of the second map and reduce phases amounts ca. 28 seconds. The reduce phase of the second run takes as long as the map phase of the second run, since the mapper is an identity mapper.

Q8 is a long-running OLAP query. The map tasks process the whole input, slightly modify it, and pass it to the reducers. Therefore, the map phase has an extremely high execution time because of processing the whole output, which is bounded by the I/O bandwidth. Further, the reducer scans the whole slightly aggregated output of the mapper and filters the data. For this reason, the execution time is only a factor 2 smaller than that of the map function. This results in an up to 12 times higher response time than that of the other queries.

Q10 checks various conditions in its selection (map) phase in order to filter the input data, which explains the bigger portion of execution time spent in the map phase. The reducers just count and group the mappers' output.

Q11 needs two map/reduce runs, which affect its run time. The first map and reduce run executes the actual work: the mapper performs as usual the selection and the reducer ensures that the correlated triples lie in the given time interval. The second map/reduce run is responsible for the grouping and counting. It takes only one fourth of the execution time compared to the first map/reduce phase.

Table 5.3 shows the average of the map and reduce execution time for each query measured for the 16-node configuration on the small, one-day data set as shown in Figure 5.5 and records the total execution time of the queries. The sum of the map and reduce function's runtime can exceed the depicted total execution time, since the reduce phase can be launched by Hadoop before the map phase is completed. This is due to the fact that Hadoop reserves system resources for the reducers in advance. As soon as the map tasks produce some output, the reducers can begin with the processing: they can fetch the data and sort it. The real processing of the data by the reducers can, however, not be proceeded until all map outputs were created. Note, that the query execution times in Table 5.3 are mainly dominated by the map tasks. The reason for this is that they have to filter the relevant triples from the huge input data and their execution time consists mainly of the I/O overhead. The reducers however get a greatly reduced input so that their execution time is only a fraction of the mappers' execution time.

When scaling the number of nodes in the cluster from 4 to 8, the query performance first increases proportionally. This is justified by the less data per node that has to be processed (the bigger overall number of map and reduce tasks). However, when we scale further from 8 to 16 cluster nodes, the query performance does not improve by the same factor, it increases only slightly. As the total number of allocated map and reduce tasks increases, there is more overhead for managing the cluster nodes. Thus, the fixed overhead increases slightly when new nodes are added to the cluster. In our case, this additional overhead is not completely compensated by the faster processing of the 16 nodes. The reason for this is the

Execution times of the map and reduce phases (sec)											
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
map	88	88	113	72	100	88	88/15	437	89	104	94/15
reduce	46	47	71	33	55	45	45/12	215	45	37	46/12
total	101	102	126	93	114	103	103/28	675	101	117	109/27

Table 5.3: The portion of execution time (in sec) consumed by the map and reduce procedures for each query on the one week data set. Q7 and Q11 consist of two map and reduce phases, the execution times are divided by a slash, respectively. The sum of both values can exceed the total execution time since the reduce phase can be launched by Hadoop before the map phase is completely finished.

following: the bigger amount of mappers increases the performance, since the input data is big enough and requires a total number of 100 mappers. On the other side, however, most of the queries (all except for Q9, Q10) do not need more than 8 reducers for executing the map outputs, i. e., the result of the hash partitioning after the map phase is divided into less than 8 groups. Therefore, applying more nodes and more reducers does not improve the query performance in this case. The performance of the 16-nodes, compared to the 8-nodes configuration is improved only in the map portion of the execution time.

Further, in order to have a fair comparison between MapReduce and our dedicated Triple Store, we reduce the size of the data read by the MapReduce tasks. Since most of the queries in our benchmark are range queries over a particular time interval, Hadoop does not need to read the whole input data, but only the data produced within the respective time interval. For an RFID scenario, one can consider that the input files are stored as small files ordered by timestamp, e. g., every day one event data file is written. Taking this into account, we can execute the time range queries only on the file log of the day containing the requested timestamps. The RFID range queries are therefore executed on an one-day-log. The experimental results can be seen in Figure 5.5. Since Q2 determines the complete historical path of an object, Q3 counts all items scanned by one reader over the whole time interval, and Q8 calculates the contaminated items related to a particular object – these queries are omitted for this experiment. All other queries are time range queries and can be executed on the data within the distinguished time interval. As expected, the execution time relation between the different queries remains the same. That is, queries Q7, Q10, and Q11 are slightly longer running than the rest. However, scaling the number of nodes in the cluster has a slightly different effect on the execution time than that in Figure 5.4. As expected, scaling the number of nodes from 4 to 8 accelerates notably the execution. In contrast to Figure 5.4, however, scaling the number of nodes from 8 to 16 increases the performance less notably than for the same nodes relationship in Figure 5.4. We believe that this smaller improvement in the performance is due to the clearly smaller size of the input. The input data size and the defined block size require the overall number of 19 mappers. Since on each node two mappers run in parallel, when we take the 16-node configuration, there are some nodes that are not busy the whole time. However, they cannot begin with the actual reduce phase execution before all nodes finish their map phase. That is why the performance of the 16-node configuration increases only slightly compared to the 8-node cluster.

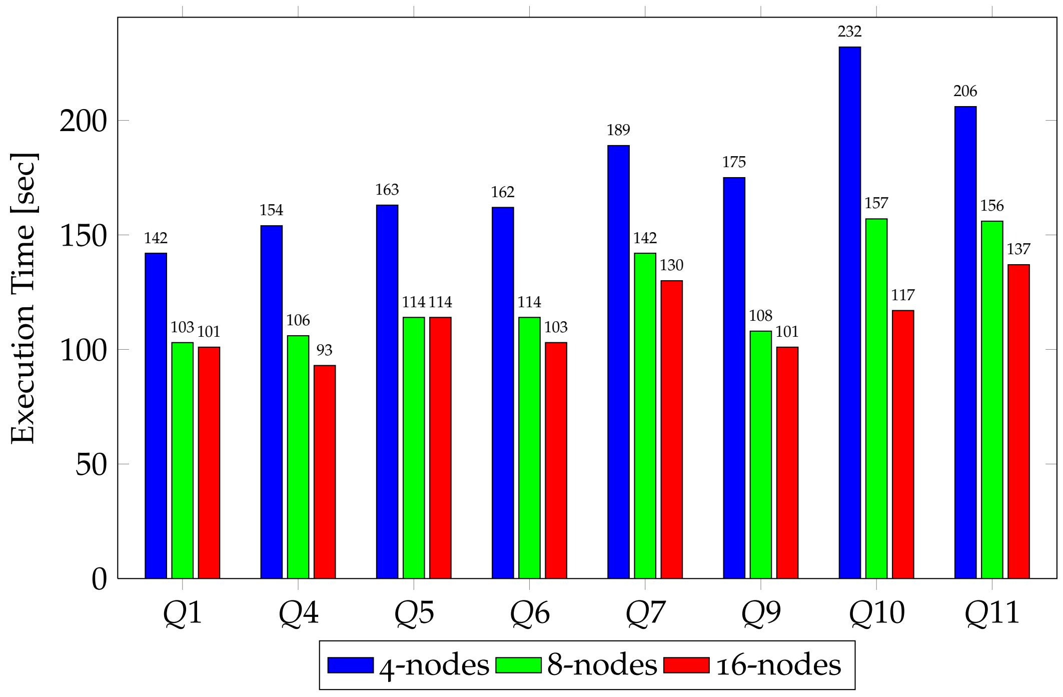


Figure 5.5: Query performance on a fraction of the data – one day data set.

As we can see, the MapReduce paradigm, in particular the Hadoop implementation, was easily adapted for an RFID use case. An interesting question is therefore where the limitations of this model are. The authors in [36] consider the distributed paradigm from a different point of view – a programming perspective. They define a programming model of MapReduce (analyzing the original Google paper [16]) based on the functional programming language Haskell and try to reveal the limitations of the system. One issue about MapReduce is that we need a good problem-specific understanding in order to determine the optimal values for the number of reduce tasks and to provide the partitioning method for the intermediate keys. In general, the MapReduce processing is restricted to the determined keys – the input as well as the intermediate results rely on keys. If we want a more general approach, that does not involve keys we have to go beyond the pure MapReduce concept and use other mechanisms like, e. g., Google's domain specific language Sawzall [42].

Overall, we can conclude that the general-purpose MapReduce approach could be applied easily for the RFID scenario, but the query performance is not optimal for our use case.

5.2 Distributed RFID Processing Using the RFID Triple Store

Hadoop is a mature and flexible approach for the distributed processing of large-scale data analysis, but it does not provide a performance that is suitable for RFID query processing. The RFID Triple Store described in Chapter 4 provides the best results in a centralized RFID scenario. Therefore, we explore the possibilities of applying this solution in a distributed manner for efficient query processing.

5.2.1 Distributed Architecture

Figure 5.6 shows the architecture of a framework for a distributed RFID processing using the RFID Triple Store. On each node of the cluster, we install an instance of the RFID Triple Store, as well as a client instance that manages the database access using a JDBC connection. The data of the different storage instances is hash partitioned by the EPC value and is nearly equally distributed over the nodes. The controller is a monitoring instance, which manages and synchronizes the parallel execution of the queries on the nodes in the cluster. It distributes the queries over the nodes and collects the query results that are then globally merged. If further processing of the results is needed (e. g., duplicate elimination), the controller takes care of it. The controller has a similar role like the Hadoop's job monitoring task.

For each query, we describe how it is implemented in our distributed scenario and whether the results have to be further processed by the controlling instance at the end.

Q1: Last location of an object

For this query, we select all events which contain the requested object ID (EPC). Since the data of the RFID Triple Store instances is partitioned by the hash value of the EPC, we send the query to the particular node in the cluster that hosts the item. No global merging of the results is needed.

Q2: The pedigree (complete path) of an object

Like for Q1, we select all events which contain the requested object ID (EPC). We therefore query only one particular RFID instance (node) and no global merging of the results is needed.

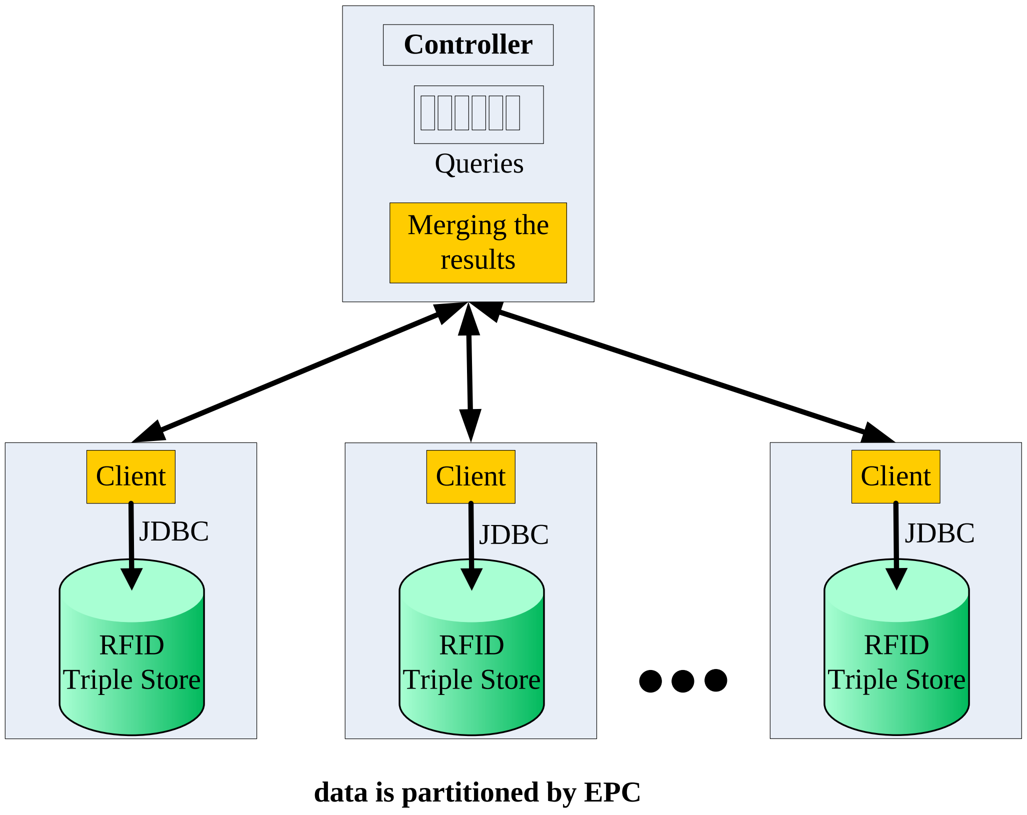


Figure 5.6: Distributed RFID processing using the RFID Triple Store.

Q3: The number of objects scanned by a certain sensor

For this query, all instances of the cluster have to be queried. Each node provides the number of events read by the particular sensor as result. At the end, the controller has to perform a global count of the sub-results from the different nodes. However, no duplicate checking is needed since no objects will be double counted due to the hash partitioning on EPC. The execution time of this query includes the time of the additional global merge step.

Q4: A list of objects scanned by a sensor within a time interval

The query is sent to all instances of the cluster. Each node delivers the unique EPCs of the items that were scanned by the sensor in the particular interval. The client has to perform a global merge of the sub-results. Because the distribution is conducted by EPC, each node will perform duplicate elimination. The same EPC cannot occur again at another node. Therefore, no global distinct operator is needed for the query. The query execution time includes the time for the results to be merged by the client.

Q5: A list of objects which were scanned by sensor rdr_1 after a time threshold and by sensor rdr_2 before a time threshold

For Q5, the query is executed by every node of the cluster and a join over the EPC value is performed. The result comprises all objects that were scanned by the two given sensors within the given time constraints without regarding the sequence in which these events passed the readers. Consider, that the sensors have a specific semantic, like entry and exit. This query determines for example the objects that passed an entry sensor after 8 o'clock in the morning and an exit sensor before 12 o'clock in the morning, i. e., all objects produced before noon. Again, at the end, the output of the different nodes has to be merged without additional processing.

Q6: A list of objects which were scanned first by sensor rdr_1 and then by sensor rdr_2 within a time interval

Q6 is executed by all nodes in the cluster. The results are globally merged by the controller without additional processing.

Q7: The number of objects which were scanned first by sensor rdr_1 and then by sensor rdr_2 within a time interval

For Q7, every node in the cluster calculates the number of objects that fulfill the select condition. They return their local counter of the result triples. The controller sums up the individual outputs and provides the total number of all objects satisfying the conditions. No global duplicate elimination is needed for the query. Additional computational time is needed at the end due to the merge step of the single counters.

Q8: A list of objects that were at sensor rdr, together with an object x (suspected to be contaminated) within a certain time interval (contamination query)

Q8 is a heavy OLAP query that has to be computed in two steps in a distributed scenario.

Step 1: we generate a subquery that returns all tuples of the form $\langle rdr, ts \rangle$ for the contaminated object. This means that we select all locations of the object with EPC x and the timestamps for these locations. This subquery is analogous to Q2 and is only submitted to one cluster node where the EPC is residing.

Step 2: For each result triple of step 1, we generate a subquery that provides all objects that were scanned by the specified reader and whose timestamp lies within the contamination interval. This subquery is executed in parallel at all nodes. The final results are therefore composed by the union of the results of all subqueries.

Q9: Listing the number of objects per reader and timestamp, which passed in a certain time interval (e.g., to identify production peaks)

For Q9, every node in the cluster must execute the query. However, in this case we cannot just merge the results of the individual nodes, but have to perform a global sum and duplicate elimination over the result set since the same reader and timestamp values can occur at every node. This additional distinct operator phase represents an additional overhead for the query execution.

Q10: Listing the number of all objects scanned by 10 readers within a time interval, grouped and ordered by reader, and a time interval of a second

Q10 is executed from every cluster node. Again, we cannot just merge the results of the individual queries, but have to perform a global distinct over the result set since the same reader and timestamp values can occur at every node. This additional distinct operator phase represents an additional overhead for the query execution.

Q11: Listing the number of all objects which were scanned within a time interval by the sensors rdr_1, rdr_2, and rdr_3 in this order, aggregated per second

For Q11, every node in the cluster must execute the query. In this case we cannot just merge the results of the individual queries, but have to perform a global duplicate elimination over the count results since the same timestamp values can occur at every node. This additional duplicate elimination phase represents an additional overhead for the query execution.

5.2.2 Performance Evaluation

In this section, we present the setup of the distributed RFID platform and discuss the achieved query performance.

Experimental Setup

For the experiments, we use the same hardware equipment as used for our Hadoop experiments in Section 5.1. The detailed description of the environment can be taken from this section. We set up a cluster of four nodes. On each node, the RFID Triple Store instance and the corresponding JDBC client are installed. One of the nodes additionally hosts the controller instance, which coordinates the execution and merges the results.

We generate the amount of RFID data produced in one week in a world-wide scenario: 450 million events. This is the same data set we used for evaluating the RFID query performance using Hadoop. The data is partitioned over the nodes using a hash function for the EPC values. This results in almost equal data load for each node (ca. 112 million events). The size of the database file for each node is 11 GB.

The queries for the RFID query workload are implemented as described above. The implementation of the distributed approach using the RFID Triple Store is prototypical, i. e., it is mainly optimized for the given workload and serves as a proof of concept. It is supposed to show that this solution performs better than a general-purpose solution like MapReduce.

Results and Discussion

The results of the query performance measurements can be seen in Figure 5.7. As expected, this approach outperforms significantly the solution using Hadoop (see Figures 5.3 and 5.4). Please note that the numbers in Figure 4.24, presenting the centralized Triple Store solution, are measured during a mixed workload consisting of concurrent queries and inserts (on a different host), so that a direct comparison with these numbers is not possible.

Query-only workload		
No.	QRT [ms]	Nodes
Q1	8	1
Q2	7	1
Q3	14	4
Q4	17	4
Q5	58	4
Q6	20	4
Q7	16	4
Q8	176	1 + 4
Q9	355	4
Q10	89	4
Q11	24	4

Figure 5.7: The average query performance on the distributed RFID platform. The last column shows the number of nodes involved in the query execution.

As shown in Figure 5.7, the OLTP queries Q1 and Q2 are both executed on one cluster node (one fourth of the original data) and have similar performance, since they both select data for a particular EPC. In the first case, the last reader that scanned the object is selected. In the second case, all readers that the object passed through are selected. In Section 4.5, the average path length of an object according to our data generation is shown. Since in our case the path length for Q2 is only

6, no significant network overhead is experienced compared to Q1.

Q3 is executed on all four nodes in parallel. At the end, the monitoring instance sums up the counter values that come as an output of the four queries. Each one of the four queries has one result tuple, similar to Q1. The higher response time of Q3 is due to the additional synchronization overhead of the four clients.

For queries Q4, Q5, and Q6, no additional post-processing of the results by the controller is needed. Each of these queries is executed on all four nodes. The reason for the higher response time of Q5 compared to the other two, is the size of its result set, which is twice as big as the result sets of Q4 and Q6.

Q7 is executed on all four nodes. For this query, similar to Q3, the counter result tuple of each query is summed up to a global counter.

Q8 is executed in two phases as described above. In the first phase, only one node is queried and its result set contains pairs $\langle rdr, ts \rangle$. From the result of this phase, the queries for the second phase are constructed. The queries of the second phase are sent to all four nodes and determine all objects that were scanned at a particular reader in the contaminated interval.

For Q9, Q10, and Q11, additional post-processing of the results is needed – a global distinct operator. This last merging step yields a bigger computational overhead that depends on the size of the result sets. The higher execution time of Q9 is explained by the big size of its result set (29900 tuples) compared to the other two queries (286 and 2 respectively).

5.3 Summary and Conclusions

In this chapter, we first applied the MapReduce paradigm for RFID data management and then we set up a distributed environment for executing queries on the RFID Triple Store.

For the management of RFID data, MapReduce provides a feasible approach as it incurs practically no overhead for storing the data. It can simply use log files containing the sensor events. Furthermore, for analytical workloads MapReduce provides good performance. However, the fixed costs of query processing heavily influenced by scanning huge parts of the data render MapReduce inappropriate for interactive workloads. To get an idea of the performance of MapReduce for our application scenario, we implemented our workload using Hadoop [10] and conducted experiments on this RFID workload. We found out that the fixed costs per query constitute 25 seconds, which is acceptable for long-running analytical tasks only. However, for transactional workloads or interactive sessions, response times of less than one second are desirable. We conclude that MapReduce is a straightforward approach for the storage of huge amounts of data for OLAP-focused applications but does not provide sufficient performance for OLTP tasks.

As a comparison, we present a prototypical distributed environment for RFID query processing using the RFID Triple Store as a backend. We distributed the data by hash partitioning it by the EPC value. Since the RFID Triple Store has the best performance for RFID query processing among the considered approaches, it can be expected that also in a distributed environment it will outperform the general-purpose Hadoop. The results of our experiments proved this assumption. The performance of the distributed variant of our RFID Triple Store is several orders of magnitude better than that of Hadoop.

6 Conclusions and Outlook

RFID is becoming a widespread adopted technology for seamlessly tracing products, possibly across a global supply chain. It provides manufacturers with up-to-date information about the position of their products and gives companies important insights in their business processes. The term real-world awareness introduced by [29] defines the process of operation on real-time data. As a result, the latest or current data is considered in business intelligence applications. In order to achieve real-world awareness in the context of RFID data, efficient mechanisms for the management of this data are needed. However, the frequently produced big amount of RFID events constitute new challenges for modern database systems. In this thesis we identified and summarized the three main challenges posed by RFID (traceability) data:

(1) The RFID sensors produce a huge amount of data per second. We estimated that for a medium-sized enterprise ca. 500 events per second are generated and for a world-wide enterprise like BMW even more than 2000 events per second should be managed. The challenge is to design an architecture that can store and query this big amount of data.

(2) The high amount of incoming data requires an efficient mechanism for processing it. RFID data has to be updated continuously. Therefore, as soon as new events arrive the data staging process should be triggered. The requirements that a data staging procedure should fulfill is to be able to insert the heavy load per second into the storage system and to provide the latest data for further processing.

(3) In order to take advantage in business planning, efficient transactional and analytical query processing should be provided. The latest RFID information should be involved in the OLAP query process. As known, fast query processing requires up-to-date indexes. The challenge in this case is to manage the trade-off between the update frequency and a reasonable query response time.

This work focused on determining possible data storage and management solutions for RFID data that fulfill the challenges.

First, we analyzed and compared existing solutions for efficient RFID data management that were implemented on existing DBMSs. We conducted a thorough qualitative and quantitative analysis of the approaches, considering the specifics of RFID data. Further, we evaluated the approaches upon an insert-only, a query-only, and a mixed workload consisting of concurrent inserts and queries (OLTP and OLAP). This way we could measure the effect of the event inserts on the runtime of the queries. The mixed workload is of particular importance in the context of RFID data, since the high event throughput must be inserted in nearly real-time by the database solutions, in order to realize the idea of the real-world awareness. Our proposed Bloom filter approach was designed to fulfill this requirement, i. e., the latest data is used for the business intelligence analysis. Its architecture combines the OLTP and the OLAP components in one system. Due to this fact, the latest scanned events are considered in the analytical reports. The Bloom filter approach achieves the event generation frequency for a medium-sized enterprise and outperforms the baseline approach with respect to the query execution time.

Second, after analyzing the existing database approaches on standard DBMSs, we developed a scalable dedicated system addressing the challenges and exploiting the RFID data specifics, the RFID Triple Store. The Triple Store is specifically designed for the requirements of RFID data. We use RFID-aware indexes for storing the data and optimize them for the expected high insert load as we pre-reserve spare slots for the incoming events. In order to speed up query processing, the RFID data is available in different permutations of the three components of a single event and is aggregated to a different level. Further, to speed up the range queries that are typical in an RFID scenario, we leverage the traceability characteristics of the data and provide an RFID-aware query engine implementation. An important optimization is the usage of the dictionary IDs rather than the real values of the timestamps. This is possible due to the fact that timestamps grow monotonically and ordering by IDs is basically ordering by value. A further optimization favors the indexes ordered by timestamp for the range queries, in order to apply start/stop conditions during the index scan and this way to prune the result as early as possible. Applying all RFID-aware architectural decisions, our Triple Store performs notably better than the existing approaches when considering the insert throughput and the query response times. It is further compliant with the requirements posed by RFID data that we stated in this thesis.

Finally, we take a look at a distributed management of RFID data. We apply the MapReduce technology in order to evaluate whether this technique is feasible for an RFID scenario. When implementing the typical RFID workload using Hadoop, we found out that this is not optimal in an RFID context due to the nature of the queries. They suffer from (1) the lack of indexes and (2) from concentrating the computation mainly in the map-functions, so that most of the reducers are idle. We further create a framework for the distributed RFID query processing using the RFID Triple Store as a storage platform. This approach outperforms the solution using Hadoop.

This thesis addressed the main challenges for efficient management of RFID data. However, there are still open issues to be solved in the context of processing RFID data. A future research topic could concentrate on exploring more possibilities for RFID event management in a distributed environment. Using the RFID Triple Store, a framework should be devised that executes not only distributed RFID queries, but also inserts new event batches concurrently.

Another future topic is to devise mechanisms for efficient extraction of old RFID data from the RFID Triple Store. Old data (e. g., data older than 3 months) do not play an important role for the daily business and for OLAP queries with a shorter foresight. Outdated events should be therefore extracted and deleted from the RFID Triple Store and archived in a storage, where they won't be accessed often.

Another topic which is often considered in the context of RFID data is data cleaning, i. e., detecting duplicate or false readings and filtering them out. There is a lot of on-going work in this field. Implementing one of the existing algorithms for data cleaning in the RFID Triple Store is subject of future work.

Bibliography

[1] MonetDB. `http://www.monetdb.org`.

[2] The DBpedia Knowledge Base. http://dbpedia.org/.

[3] W3C: Resource Description Framework (RDF). `http://www.w3.org/RDF/`.

[4] W3C: SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[5] R. Agrawal, A. Cheung, K. Kailing, and S. Schönauer. Towards Traceability across Sovereign, Distributed RFID Databases. In *IDEAS*, pages 174–184, 2006.

[6] Y. Bai, F. Wang, and P. Liu. Efficiently Filtering RFID Data Streams. In *CleanDB*, 2006.

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD Conference*, pages 283–296, 2009.

[8] BMW. Quarterly Report to 30 September 2009. `http://www.bmwgroup.com`. accessed February 19, 2010.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[10] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[11] R. Brunel. Adapting the RDF-3X System for the Management of RFID Data. Bachelor's Thesis. Supervised by Veneta Dobreva and Martina Albutiu, Fakultät für Informatik, Technische Universität München, 2010.

[12] Z. Cao, C. Sutton, Y. Diao, and P. Shenoy. Distributed Inference and Query Processing for RFID Tracking and Monitoring. In *VLDB*, pages 326–337, 2011.

[13] H. Chen, W.-S. Ku, H. Wang, and M.-T. Sun. Leveraging Spatio-temporal Redundancy for RFID Data Cleansing. In *SIGMOD Conference*, pages 51–62, 2010.

[14] J. Collins. Boeing Outlines Tagging Timetable. *RFID Journal*.

[15] J. Collins. DOD Tries Tags That Phone Home. *RFID Journal*.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[17] J. Dean and S. Ghemawat. MapReduce: a Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, 2010.

[18] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):518–529, 2010.

[19] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.

[20] V. Dobreva and M.-C. Albutiu. Put All Eggs in One Basket: an OLTP and OLAP Database Approach for Traceability Data. In *IDAR '10: Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research*, pages 31–36, New York, NY, USA, 2010. ACM.

[21] V. Dobreva, M.-C. Albutiu, R. Brunel, T. Neumann, and A. Kemper. Get Tracked: A Triple Store for RFID Traceability Data. In *ADBIS*, pages 167–180, 2012.

[22] A. Eickler, C. A. Gerlhof, and D. Kossmann. A Performance Evaluation of OID Mapping Techniques. In *VLDB*, pages 18–29, 1995.

[23] EPCGlobal: EPC Tag Data Standards Version 1.4, Ratified Specification. `http://www.epcglobalinc.org/standards/`, June 2008.

[24] R. B. Ferguson. Logan Airport to Demonstrate Baggage, Passenger RFID Tracking. *eWeek*.

[25] K. Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. Wiley Publishing, 2003.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, pages 29–43, 2003.

[27] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *ICDE*, page 83, 2006.

[28] P. Harrop. RFID in the Postal Service. *MoreRFID*.

[29] C. Heinrich. *RFID and Beyond: Growing Your Business Through Real World Awareness*. Wiley Publishing, 2005.

[30] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. In *VLDB*, 2011.

[31] S. R. Jeffery, M. N. Garofalakis, and M. J. Franklin. Adaptive Cleaning for RFID Data Streams. In *VLDB*, pages 163–174, 2006.

[32] JENA. Jena - A Semantic Web Framework for Java. `http://jena.sourceforge.net/`.

[33] S. Kinauer. Applying MapReduce for RFID Data Management. Bachelor's Thesis. Supervised by Veneta Dobreva and Martina Albutiu, Fakultät für Informatik, Technische Universität München, 2010.

[34] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115, 2004.

[35] S. Krompass, S. Aulbach, and A. Kemper. Data Staging for OLAP- and OLTP-Applications on RFID Data. In *Database Systems for Business, Technology, and Web (BTW)*, pages 542–561, 2007.

[36] R. Lämmel. Google's MapReduce Programming Model - Revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[37] C.-H. Lee and C.-W. Chung. Efficient Storage Scheme and Query Processing for Supply Chain Management using RFID. In *SIGMOD Conference*, pages 291–302, 2008.

[38] E. Masciari. RFID Data Management for Effective Objects Tracking. In *SAC*, pages 457–461, 2007.

[39] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD Conference*, pages 627–640, 2009.

[40] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB J.*, 19(1):91–113, 2010.

[41] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[42] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[43] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-memory Column Database. In *SIGMOD '09: Proc. of the 35th SIGMOD Intl. Conf. on Management of data*, pages 1–2, New York, NY, USA, 2009. ACM.

[44] E. Prud'Hommeaux and A. Seaborne. SPARQL Query Language for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115, January 2008.

[45] RFID Journal. Dedicated to Radio Frequency Identification and its Business Applications. Vol.5, No.4.

[46] K. Rosen. *Elementary Number Theory: And Its Applications*. Addison-Wesley, 2011.

[47] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan. MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR). In *MSR*, pages 21–30, 2009.

[48] C. Sosnowski. Handling RFID Data in Databases. Diploma thesis, Fakultät für Informatik, Technische Universität München, 2006.

[49] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[50] C. Swedberg. Hospital Uses RFID for Surgical Patients. *RFID Journal*.

[51] J. Tang, J. Sun, C. Wang, and Z. Yang. Social Influence Analysis in Large-scale Networks. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 807–816, New York, NY, USA, 2009. ACM.

[52] N. Tatbul. Streaming Data Integration: Challenges and Opportunities. In *IEEE ICDE International Workshop on New Trends in Information Integration (NTII'10)*, Long Beach, CA, March 2010.

[53] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[54] K. Valev. Implementation and Evaluation of Existing Approaches for the Management of RFID Data. Bachelor's Thesis. Supervised by Veneta Dobreva and Martina Albutiu, Fakultät für Informatik, Technische Universität München, 2011.

[55] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1):1008–1019, 2008.

[56] R.-M. Wernicke. Entwicklung eines Frameworks zur Evaluierung existierender Ansätze für das Management von RFID Daten. Master's Thesis. Supervised by Veneta Dobreva and Martina Albutiu, Fakultät für Informatik, Technische Universität München, 2011.

[57] X. Wu, M.-L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE*, pages 66–78, 2004.