

12. GI-Fachtagung für Datenbanksysteme in
Business, Technologie und Web (BTW 2007)
5. bis 9. März 2007 - Aachen, Germany
<http://www.btw2007.de/>

Data Staging for OLAP- and OLTP-Applications on RFID Data

Stefan Krompaß, Stefan Aulbach, Alfons Kemper

Technische Universität München – Lehrstuhl für Informatik III
85748 Garching bei München
(krompass|aulbach|kemper)@in.tum.de

Abstract:

The emerging trend towards seamless monitoring of all business processes via comprehensive sensor networks – in particular RFID readers – creates new data management challenges. In addition to handling the huge volume of data generated by these sensor networks the information systems must support the efficient querying and analysis of both recent data and historic sensor readings. In this paper, we devise and evaluate a data staging architecture that consists of a distributed caching layer and a data warehouse. The caches maintain the most recent RFID events (i.e., sensor readings) to facilitate very efficient OLTP processing. Aged RFID events are removed from the caches and propagated into the data warehouse where related events are aggregated to reduce storage space consumption. The data warehouse can be utilized for business intelligence applications that, e.g., analyze the supply chain quality.

1 Introduction

Radio frequency identification (RFID) is expected to become the key technology for monitoring object movement by storing product information on tags which are attached to individual items. RFID sensors (or RFID readers) can read the identification of an item without requiring either contact or line of sight. Thus, with sensors placed at various locations, the movement of items can be tracked. For this purpose the electronic product code (EPC, [EPC06]), an identification scheme for the real-time, automatic identification of objects, has been standardized by EPCGlobal [Hei05].

From a database perspective, the challenge is to manage the data to enable comprehensive control and analysis. There are two types of applications which are important for the analysis: On the one hand, lightweight queries are processed on most recent data, i.e., information associated with the current location and the last time an item was scanned. These queries are submitted frequently, e.g., by an object tracking service where customers can query the current location of their order or by an RFID-based process control ([TDF06]). On the other hand, OLAP queries allow business intelligence applications to analyze, for example, the quality of supply chains. The OLAP queries on the warehouse are less frequently issued in comparison to the lightweight queries, which are run against the caches, but process a much larger data volume.

In this paper, we devise and evaluate a data staging architecture that consists of a distributed caching layer and a data warehouse. The caches constitute main-memory databases which maintain the most recent RFID events to facilitate very efficient query processing.

Aged RFID events are removed from the caches and are propagated into the data warehouse where related events are aggregated to reduce storage space consumption. Currency of data ([CKRS04]) is an integral part of our novel architecture. In contrast to traditional warehouses, where it is acceptable to have updates only infrequently and at predictable times, our data staging approach updates the warehouse on the fly. For building such an RFID infrastructure, the contributions of this paper can be summarized as follows.

- **Database design.** Our proposed database schema efficiently precomputes and stores the path information of the objects in the warehouse.
- **Data cleaning.** In our architecture, a cache maintains the most recent data about an item. But not all events are necessary for maintaining the aggregated data in the warehouse. The cache filters these events and propagates only information relevant for business intelligence applications to the warehouse.
- **Data staging.** We describe two novel data staging mechanisms, *tuple-wise data staging* and *bulk data staging* to update the aggregated data in the warehouse. The data staging process is only triggered when tracked objects move to another location. While the *tuple-wise* approach updates the warehouse on a state change for individual items, the *bulk* approach processes state changes for groups of items.
- **Validation.** We present analytical estimates and benchmarks which show that traditional database schemas are not sufficient for tracking moving objects. Furthermore, we demonstrate the impact of the data cleaning and data staging mechanisms on the performance of RFID event processing.

The rest of the paper is organized as follows. Section 2 gives an overview of our RFID architecture. In Section 3, we explain how RFID events are processed. We give a description of the data staging and explain our two approaches for data staging: *tuple-wise data staging* and *bulk data staging*. Benchmarks for the quantitative evaluation of our architecture are presented in Section 4, followed by an overview of related work in Section 5. In Section 6, we summarize our studies and outline ongoing and future work on this subject.

2 Architecture

This section gives an overview of the components and how they are connected to each other before the design details of the cache and the warehouse are presented.

2.1 System Design

The Item Tracking System is implemented as a 3-tier-architecture, consisting of data stores at the lowest level, middleware atop of it and finally a set of clients, which can be either sensors or query clients. Figure 1 shows a comprehensive overview.

Data Stores The data stores are divided in two main storage areas, depending on the kind of data they store. The different data stores reflect the different types of queries a client can execute. Transactional processing takes place within the cache, while analytic processing is done within the warehouse.

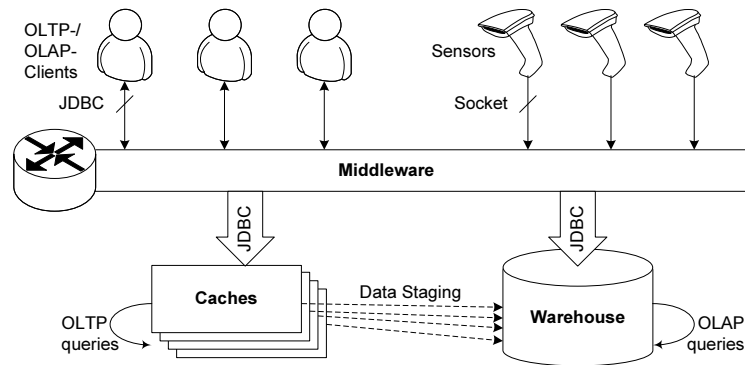


Figure 1: Architecture

The first data store type is the cache layer, which stores the most recent data inside a main memory relational database system used as cache. After receiving data from a sensor, the cache is updated to reflect the current item state. The middleware keeps track of the changes to the warehouse and ensures that no information necessary for constructing the warehouse data is overwritten.

This paper illustrates the usage of just one cache for the most recent data, but ongoing work is focused on scaling the system to multiple caches. Multiple caches are combined into a *cache group*, which in fact constitutes a partitioned database. Every cache group contains all items that are known to the system. In order to facilitate different query types, multiple cache groups can be set up, but just one cache group is necessary for data staging. In such a case, the information about item locations is stored redundantly.

The partitioning is based on different strategies. For example, a single RFID sensor with a very high scan rate (e.g., at the entrance of a warehouse) may have its own cache, while the other sensors share the remaining caches. We denote this partitioning strategy as *reader-based*. The other type of strategy is *item-based*, where the partitioning is based on product groups, supplier or other item-specific attributes. Combining multiple strategies leads to *redundant* data caching.

The second data store type is the data warehouse. The warehouse is based on a conventional relational database management system and contains historic data needed to analyze the sensor readings, e.g., for analyzing the supply chain over a longer period of time. Section 3 describes the data staging algorithms for propagating recent cache data into the warehouse.

Middleware The middleware is responsible for data cleaning, to compress and optimize the raw RFID data streams generated by the sensors. Furthermore the middleware triggers the data staging. Placing this functionality into the middleware enables to intercept cache update operations that may lead to overwritten data, and initiate a data staging first.

Different middleware plug-ins support a variety of interfaces to sensors, clients and even data staging algorithms. Currently, two data staging algorithms are implemented that will be compared in the course of this paper. As part of the data staging procedure, the middle-

RFID_RAW	
item	The EPC of the item
reader	The EPC of the reader that scanned the item
t_in	The time when the item arrived at the current path
timestamp	The last timestamp when the item was scanned by the reader
subpath_out	The time when the item was last scanned by the previous reader

Table 1: Column Description of Cache Table RFID_RAW

RFID_RAW				
item	reader	t_in	timestamp	subpath_out
i_1	s_3	t_6	t_6	<i>null</i>
i_2	s_3	t_6	t_6	<i>null</i>
i_3	s_4	t_5	t_6	<i>null</i>
i_4	s_4	t_5	t_6	<i>null</i>
i_5	s_5	t_5	t_6	<i>null</i>
i_6	s_5	t_5	t_6	<i>null</i>

Table 2: Example of Cache Contents After Scanning at Timestamp t_6

ware is responsible for cleaning the cache to avoid cache overflow.

Sensors and Query Clients Two types of clients use the interfaces provided by the middleware. On the one hand, data sensors acquire data and send it to the middleware via socket connections. Each sensor produces *RFID events* ($epc, reader, t$), where epc represents a scanned item, $reader$ is a unique identifier of the reader and t is the time when the item passed the sensor area. The interface to the sensors can be extended in future, e.g., to support environmental sensors in refrigerated stores, even integrated in RFID tags ([OTS⁺06]). The socket-based connection allows the exploitation of modern, high performance network technologies (e. g. the *Socket Direct Protocol* (SDP) via InfiniBand) in order to decrease the protocol overhead.

On the other hand, there are clients querying the data stores. OLTP queries that rely on the most recent data are processed by the cache, while OLAP queries on historic data are processed by the warehouse. The middleware acts as a proxy for the clients so that incoming queries via JDBC are redirected to the appropriate data store(s).

2.2 Cache Design

The first part of the database schema consists of the table RFID_RAW that is maintained in the cache. This relation, shown in Table 1, contains the most recent RFID entries – cleaned and processed by the middleware. As part of the data cleaning, a new entry is inserted into RFID_RAW if a yet unknown item has been scanned. Otherwise, the existing entry of that particular scanned item is updated. The detailed description of the procedure of inserting events into the cache is part of Section 3. An example set of such entries is given in Table 2.

2.3 Data Warehouse Design

Every item travels along a *path*, consisting of the RFID readers the item passes by. For each reader on the path, the warehouse stores the timestamps when an item arrives at this reader and leaves the reader, respectively. Since the RFID_RAW relation inside the cache is independent of the contents of the warehouse, different approaches for representing this path information can be used.

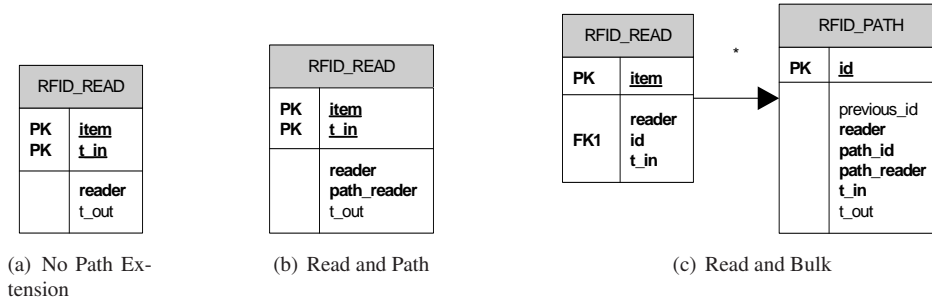


Figure 2: Different Schemas Used Inside the Warehouse

2.3.1 Single Event Approach: *No Path Extensions*

With the single event approach, every location change of an item is stored in a single row in the table `RFID_READ`. An entry in this table contains the times when the item reaches and leaves the reader in columns `t_in` and `t_out`. Figure 2(a) shows the schema of the single event approach. Table 3 contains some example entries ¹.

During data staging no additional processing has to be done. The only task is to insert new rows into the table `RFID_READ`, so there is no need for any space or time consuming transformation, and the data staging run is quite fast. However, the approach incurs some severe drawbacks: since no path information is stored, this information must be computed at query time, which significantly impacts the query performance.

If a client wants to reconstruct the path of a certain item, multiple rows must be fetched. The number of rows corresponds to the path length. As the events are inserted sequentially, the path can be reconstructed by retrieving all events of a single item sorted by the column `t_in`.

2.3.2 Path Materialization Approach: *Read and Path*

The query performance analysis of the previous approach leads to an improvement by maintaining an additional attribute `path_reader` in the `RFID_READ` relation as depicted in Figure 2(b). The column `path_reader` contains the path the item has traversed until it has reached the current reader. To obtain the path of a certain item, only the row corresponding to its most recently processed event must be returned.

Expediting the query processing induces a slightly more expensive processing during data staging, but our performance evaluation in Section 4.1 indicates that this data staging overhead is tolerable. However, a major problem remains: Maintaining the path information in every tuple induces an intolerable space consumption. Table 3 shows example entries based on this schema.

¹This table contains an additional attribute `path_reader`, which is used by the following data schema and is not part of the *No Path Extension* schema.

RFID_READ				
<u>item</u>	<u>t_in</u>	<u>t_out</u>	<u>reader</u>	<u>path_reader</u>
i_1	t_{11}	t_{12}	r_1	" r_1 "
i_1	t_{21}	t_{22}	r_2	" $r_1; r_2$ "
i_1	t_{31}	<i>null</i>	r_3	" $r_1; r_2; r_3$ "

Table 3: Example Data for Data Schema *Read and Path*

RFID_READ	
item	The EPC of the item
reader	The EPC of the reader that scanned the item
id	The id of the path on which the respective item currently is
t_in	The time when the item arrived at the current path

Table 4: Column Description of Item Information Table `RFID_READ`

2.3.3 Bulk Movements Approach: *Read and Bulk*

To increase the efficiency of the business analytics process, we extend the schema to incorporate the observation according to [GHLK06] that RFID tagged items generally travel in *bulks*, i.e., groups along the same path. The main modification compared to the previous approaches consists of the additional table `RFID_PATH` which maintains the path information without redundancy. The path information is referenced from the `RFID_READ` table. As we consider groups of multiple items that reference the same path, the space consumption is reduced compared to the previous approach. As depicted in Figure 2(c) an entry in the `RFID_READ` table consists of the last reader that scanned the item, a reference to the path the item has traversed and the timestamp when the item entered the scan area of the last reader. Table 4 explains the columns in detail and Table 6 shows some sample entries.

The path information is stored in `RFID_PATH`. For every identified path a set of entries is generated in this table. Since many items travel along the same route, those items can share a path which is composed of sub-paths that model the situation when an item stayed at a certain reader. These sub-paths are connected via the `previous_id`; if an entry describes the start of a path, the `previous_id` is undefined. The construction of these paths is covered in the next section. Table 5 describes the columns of this table and Table 7 illustrates a sample configuration.

RFID_PATH	
id	The identification of the path
previous_id	The identification of the previous sub-path
reader	The reader that scanned the item most recently
path_id	The string representation of the ids of sub-paths of this path
path_reader	The string representation of the readers which scanned the items
t_in	The time when the current path was reached (i.e., was first scanned by the current reader)
t_out	The time when the first item of the group left the range of the respective reader

Table 5: Column Description of Path Information Table `RFID_PATH`

RFID_READ			
item	reader	id	t.in
i_1	s_3	p_8	t_6
i_2	s_3	p_8	t_6
i_3	s_4	p_5	t_5
i_4	s_4	p_5	t_5
i_5	s_5	p_6	t_5
i_6	s_5	p_6	t_5

Table 6: Example Item Information in the Warehouse

RFID_PATH						
id	previous.id	reader	path.id	path.reader	t.in	t.out
p_1	<i>null</i>	s_1	" p_1 "	" s_1 "	t_1	t_2
p_2	p_1	s_4	" $p_1; p_2$ "	" $s_1; s_4$ "	t_3	t_4
p_3	<i>null</i>	s_1	" p_3 "	" s_1 "	t_1	t_3
p_4	p_3	s_2	" $p_3; p_4$ "	" $s_1; s_2$ "	t_4	t_4
p_5	p_4	s_4	" $p_3; p_4; p_5$ "	" $s_1; s_2; s_4$ "	t_5	<i>null</i>
p_6	p_2	s_5	" $p_1; p_2; p_6$ "	" $s_1; s_4; s_5$ "	t_5	<i>null</i>
p_7	p_3	s_2	" $p_3; p_7$ "	" $s_1; s_2$ "	t_4	t_5
p_8	p_7	s_3	" $p_3; p_7; p_8$ "	" $s_1; s_2; s_3$ "	t_6	<i>null</i>

Table 7: Example Path Information in the Warehouse

As we will see in the comparison of the schemas in Section 4, this schema essentially increases the effectiveness of queries and reduces the amount of used disk space, but the user has to cope with a more complex database design, and the data staging runs consume more time.

3 Event Processing

An incoming event (e, s, t) is first processed by the cache to maintain the most recent data. If necessary, RFID events are further processed to update the warehouse.

The event processing described in this section is based on two assumptions. First, we assume reliable RFID data streams, i.e., none of the RFID events generated by the sensors is dropped. Dealing with unreliable RFID streams is not in the scope of this paper, but is addressed in, e.g., [JGF06]. Second, the timestamps of the incoming events are increasing, i.e., "older" events cannot overtake "younger" events with smaller timestamps.

3.1 Event Processing in the Cache

For each item, the cache stores the current location and the corresponding time information in the table `RFID_RAW` (Table 1). The current location of the item is represented by the sensor s that most recently scanned the item. The time information comprises times t^F and t^R the item was first and most recently scanned by s , as well as time t^L when the item was last scanned by the previous sensor.

For each incoming event (e, s, t) the information in the cache is updated, as shown in the algorithm in Figure 3. If a new item has been scanned, i.e., EPC e is not yet stored in the cache, the algorithm inserts a new tuple for the item which stores the current sensor s of the item. Furthermore, as the item was first scanned by s , t^F and t^R are initialized with the incoming timestamp t . As the item has not been scanned by any other sensor, t^L does not need to be initialized. Thus, the resulting tuple for item e in table `RFID_RAW` is:

$$\langle e, s, t, t, null \rangle$$

If there is an entry for e in `RFID_RAW`, the algorithm checks if the item moved to another sensor. Let s_{cur} denote the sensor currently stored for item e in the cache. If s equals s_{cur} , the item did not leave the range of the sensor. In this case, the algorithm only updates

Input: A RFID event (e, s, t) , where e denotes the EPC of the scanned item, s the sensor that scanned the item, and t the timestamp when the item was scanned by the sensor.

1. If the item is not yet stored in the cache, insert a new tuple in table `RFID_RAW` to store the current location of the item and the time when it has been scanned. No further processing is needed.
 2. Otherwise, there is an entry for e in the cache. Let s_{cur} denote the sensor currently stored for the item.
 3. Check if the item moved to a new sensor, i.e., if sensor s is different from s_{cur} .
 - (a) If item e stayed at the same reader, update the `timestamp` column for the item by setting it to t .
 - (b) If, on the other hand, e moved to another reader, store the value for e 's current `timestamp` value in `subpath_out`. Also, update the information about the sensor and the time when the item was first scanned.
-

Figure 3: Update of the Cache

e 's cache entry so that it contains the time when the item was scanned last by updating `timestamp` to t .

If the sensor s of the incoming RFID event is not equal to s_{cur} , e reached the range of a new sensor. In order to allow data staging (described in Section 3.2), the last scan time at the previous sensor s_{cur} has to be stored in `subpath_out`. Let t' denote the time currently stored in `timestamp`, i.e., the time when e was last scanned by s_{cur} . Time t' is copied to `subpath_out`. After that, the sensor information including the scan time is updated:

$$\langle e, s, t, t, t' \rangle$$

In order to prevent indefinite cache growth due to continuous event generation by the sensors, we provide a mechanism, called *cache cleaning* to control the cache size. Due to space limitations, we cannot describe cache cleaning in full detail. The idea of the approach is to keep only those items in the cache that have been read within a certain time window. The information of outdated items is stored in the warehouse and can be restored if the item is scanned again.

3.2 Data Staging

The data staging process creates an aggregated view of the path information in the data warehouse. Items may travel in *bulks* that are defined by their borders. Each bulk is a group of items in `RFID_RAW`, that correspond in their current sensor s_i , in their initial appearance timestamp t_i^F at this sensor, in their most recent scanning timestamp t_i , and in their path which they have traversed. We denote the first item of a bulk as *pioneer item*. This item serves as an identifier for the whole bulk. Since the number of items and the scan order of the items inside the bulk is not fixed, the pioneer item can vary from location to location, but the varying pioneer items still identify the same bulk. Figure 4 and Table 8 show how bulks can move: Every rectangle denotes a certain bulk at the given location s_1 through s_5 .

The *path* of a single bulk is a singly-linked list of path nodes. Every path node p_i stores a

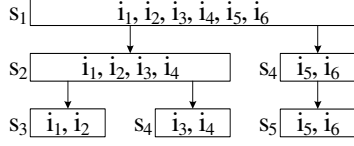


Figure 4: Movement of Objects

	s_1	s_2	s_3	s_4	s_5
i_1	t_1, t_2, t_3	t_4, t_5	t_6	–	–
i_2	t_1, t_2, t_3	t_4, t_5	t_6	–	–
i_3	t_1, t_2, t_3	t_4	–	t_5, t_6	–
i_4	t_1, t_2, t_3	t_4	–	t_5, t_6	–
i_5	t_1, t_2	–	–	t_3, t_4	t_5, t_6
i_6	t_1, t_2	–	–	t_3, t_4	t_5, t_6

Table 8: Scan Times per Item

pointer to the previous path p_{i-1} , an RFID sensor s_i , and two timestamps t_i^F and t_i^L , when a bulk was first and last scanned by s_i . Furthermore, we store for each path p_i a string representation ids_i of the path ids “ $p_1; \dots; p_i$ ” and the sensors $sens_i = “s_1; \dots; s_i”$ on the path.

The backlink of an RFID path’s starting point p_1 is undefined (*null*), thus, the entry for the first step on a path is $\langle p_1, null, s_1, “p_1”, “s_1”, t_1^F, t_1^L \rangle$. The n -th entry of a path is²

$$\langle p_n, p_{n-1}, s_n, ids_{n-1} \oplus “p_n”, sens_{n-1} \oplus “s_n”, t_n^F, t_n^L \rangle$$

If a bulk of items e_1, \dots, e_k moves from sensor s_{n-1} to s_n , the path information in table RFID_PATH has to be generated only for the pioneer item e_1 . The path information is reused for all other items e_2, \dots, e_k of that bulk.

To reduce space consumption of the path information, different bulks can share path nodes if they are identical, i.e., if the bulks initially travel along the same route and then split up. So, the overall structure regarding all bulks can be viewed as a forest.

We present two different data staging algorithms which differ in the time when they are triggered. *Tuple-wise data staging*, which is described in Section 3.2.1, is triggered each time an incoming RFID event at the cache indicates that the corresponding item moved to a new location. In contrast to that, *bulk data staging*, described in Section 3.2.2, updates the warehouse for several items in a batch.

3.2.1 Tuple-wise Data Staging

With *tuple-wise data staging*, the information in the tables RFID_PATH in the cache and RFID_READ in the warehouse are asynchronously updated if item e is scanned for the first time or if it reaches the range of a new reader. The input parameter for the data staging process is a single tuple from the cache table RFID_RAW, containing, amongst others, the time t_{n-1}^L when e left the last reader and time t_n^F when the item reached the range of s_n .

An item that is scanned for the first time cannot have a time information about when it left the previous reader. Thus, if t_0^L is not set, we can identify an item that is scanned for the first time. In this case, we generate a new path p_1 that has no predecessor. Additionally, we store the sensor s_1 and the time when the item is scanned by the sensor. Since e is on the first step of its path, “ p_1 ” and “ s_1 ” constitute the string of paths and the string of passed sensors, respectively.

$$\langle p_1, null, s_1, “p_1”, “s_1”, t_1^F, null \rangle$$

² \oplus denotes string concatenation with a delimiter “;”

Input: Single tuple $\langle \text{item} : e, \text{reader} : s_n, \text{t.in} : t_n^F, \text{timestamp} : t_n, \text{subpath.out} : t_{n-1}^L \rangle$, transferred from `RFID_RAW`, where e denotes the EPC of the scanned item, s_n the sensor that scanned e , t_n^F the time when e reached the range of s_n , t_n the timestamp, when s_n scanned e most recently, and t_{n-1}^L the time when the previous sensor last scanned e .

```

IF ( $t_{n-1}^L = \text{null}$ ) THEN (Item scanned for the first time)
    Generate new path  $p_1 : \langle p_1, \text{null}, s_1, "p_1", s_1, t_1^F, \text{null} \rangle$ 
    Insert new entry for  $e$  with a reference to  $p_1$  in RFID_READ.
ELSE (The item is scanned by a new sensor.)
    1. Retrieve  $e$ 's path  $p_{n-1}$ , currently referenced in RFID_READ's attribute id.
    (★) 2. Terminate paths:
        Let  $p^{\text{TERM}}$  denote the terminated path.
        IF ( $p_{n-1}.\text{t.out} = \text{null}$ ) THEN
             $p_{n-1}.\text{t.out} := t_{n-1}^L$ ;  $p^{\text{TERM}} := p_{n-1}$ 
        ELSE IF ( $p_{n-1}.\text{t.out} = t_{n-1}^L$ ) THEN
             $p^{\text{TERM}} := p_{n-1}$ 
        ELSE
            (a) Check, if there is already a path  $p^{\text{TERM}}$  with previous.id =
                 $p_{n-1}.\text{previous.id}$ , reader =  $p_{n-1}.\text{reader}$ , t.in =  $p_{n-1}.\text{t.in}$ ,
                and t.out =  $t_{n-1}^L$ .
            (b) If such a path exists, reuse this path as  $p^{\text{TERM}}$ .
            (c) Otherwise, if such a path does not exist, create  $p^{\text{TERM}}$  by copying  $p_{n-1}$ .
                 $p^{\text{TERM}}.\text{t.out} := t_{n-1}^L$ 
    3. For the pioneer item, create the path  $\langle p_n, p^{\text{TERM}}, s_n, \text{id}_{s_n}, \text{sens}_n, t_n^F, \text{null} \rangle$ , for all
        other items in the bulk reference the pioneer item's path.
    4. Update  $e$ 's entry in RFID_READ, set id =  $p_n$  and t.in =  $t_n^F$ .

```

Figure 5: Tuple-wise Data Staging

If the item moved from a sensor to a new sensor s_n , the information when e left the path currently stored in `RFID_READ` has to be maintained. An item that moved from one location to another is identified by a value for t_{n-1}^L which is not equal to `null`. This timestamp is used to terminate the current path of the item, i.e., the timeout information of e 's current path must be updated to contain t_{n-1}^L , before we can actually create p_n . Let p^{TERM} denote the terminated path. The update of the timeout information of paths depends on how bulks are split. Consider a bulk B at sensor s_{n-1} which splits into two bulks B_1 and B_2 . There are three cases for the termination of a path.

First, e is the first item of bulk B to be scanned by its new sensor s_n . In this case, the timeout information `t.out` in p_{n-1} is not yet set because e is the first item that leaves the range of s . For this purpose, we update `t.out` to t_{n-1}^L .

Second, consider a bulk B that splits into two bulks B_1 and B_2 . All items of B leave the range of s_{n-1} in parallel at time t_{n-1}^L . Furthermore, assume that at least one item in B_1 has already been processed as described above, i.e., p_{n-1} 's timeout value is t_{n-1}^L . Let e denote the first item of bulk B_2 to be processed. In this case, the timeout information in

path p_{n-1} is already correct for e , there is nothing more to do for path termination.

Third, in contrast to case two, the items in bulk B_2 stay longer at sensor s_{n-1} than the items in B_1 which left the range of s_{n-1} at time $\hat{t}_{n-1} < t_{n-1}^L$, i.e., p_{n-1} 's timeout information is \hat{t}_{n-1} . Let e denote the first item of B_2 to reach a new sensor. In this case, we look for a path p^{TERM} which stores the same information as p_{n-1} , but with correct timeout information. If no such path exists, we create p^{TERM} by copying p_{n-1} and setting the timeout information of p^{TERM} to t_{n-1}^L .

After terminating the path for e , the movement of e to the new sensor s_n has to be stored. For this, a new path p_n with p^{TERM} as predecessor is created. In order to build the string of path identifiers ids_n , we concatenate ids^{TERM} (string containing all paths from p_1 to p^{TERM}) and " p_n ". Similarly, the string of sensors $sens_n$ passed by the item is the concatenation of the $sens^{\text{TERM}}$ (string of sensors from s_1 to s^{TERM}) and " s_n ". Furthermore, p_n stores the sensor s_n and the respective scan time t_n^F :

$$\langle p_n, p^{\text{TERM}}, s_n, ids^{\text{TERM}} \oplus "p_n", sens^{\text{TERM}} \oplus "s_n", t_n^F, null \rangle$$

In order to link the newly generated path to the item, the algorithm updates the item information for e in `RFID_READ` to reference the newly generated path p_n :

$$\langle e, s_n, p_n, t_n^F \rangle$$

The algorithm for the tuple-wise data staging is shown in Figure 5.

Example An initial bulk of six items is successively split resulting in three bulks of two items that move from sensor to sensor. The path of the items is depicted in Figure 4 and the corresponding timestamps, when the sensors scan the items, are listed in Table 8. At the beginning, the tables in the cache and in the warehouse are empty. The schemas of the warehouse tables are depicted in Table 4 for `RFID_READ` and in Table 5 for `RFID_PATH`. For brevity, we omit the description for items that do not reach a new sensor, where only the information in the cache is updated.

Time t_1 : Data staging is triggered after the first incoming event $\langle i_1, s_1, t_1 \rangle$ (item i_1 is scanned by sensor s_1 at time t_1) is inserted into the cache. The warehouse does not yet contain any information about items, so that we insert path $\langle p_1, null, s_1, "p_1", "s_1", t_1, null \rangle$ into `RFID_PATH`. Additionally, we insert a new entry for e_1 which references path p_1 into `RFID_READ`. All other items scanned at t_1 are on the same path as i_1 . When the corresponding events for i_2, \dots, i_6 arrive, the cache is updated and data staging is triggered. As the suitable path already exists, we only insert the corresponding entries in `RFID_READ` with references to p_1 .

Time t_2 : (no data staging)

Time t_3 : Data staging is performed for i_5 which moves along with i_6 to sensor s_4 . Since i_5 is the first item scanned by s_4 , it is identified as pioneer item. The current path of i_5 , p_1 , does not store a value for timeout information. Thus, we update t_{out} to contain t_2 . After creating a new path p_2 with predecessor p_1 to reflect i_5 's movement to a new sensor, `RFID_READ` is updated to reference p_2 . For i_6 , data staging only updates the corresponding entry in `RFID_READ` to reference p_2 .

Time t_4 : The cache is updated when i_1 moves to sensor s_2 . During data staging, i_1 is identified as pioneer item because it is the first item scanned by s_2 . The timeout information for i_1 's current path p_1 contains time t_2 (data staging for i_5 at time t_3). In order to store the correct timeout information, we create a duplicate path p_3 of p_1 with correct timeout information. The newly generated path is predecessor for p_4 , which stores i_1 's movement to s_2 . The item information of i_1 in `RFID_READ` is updated to reference path p_4 . Data staging for items i_2, \dots, i_4 updates the respective entries in `RFID_READ` to reference p_4 .

Time t_5 : The arrival of i_3 at sensor s_4 triggers data staging. The timeout information of i_3 's current path is set to t_4 (the last time when i_5 was scanned by its previous reader). We use the terminated path as predecessor for path p_5 which is used to update `RFID_READ` for i_3 and, finally, i_4 . When i_5 moves to sensor s_5 at the same time, path p_2 is terminated. We create a new path p_6 which references p_2 as predecessor. The entry for i_5 in `RFID_READ` is updated to reference p_5 . For item i_6 , the path information can be reused.

Time t_6 : When i_1 and i_2 reach sensor s_3 , i_1 's current path p_4 already has a timeout value t_4 , which is different from the timestamp t_5 when i_1 was last scanned by s_2 . Thus, we duplicate p_4 by creating p_7 which contains t_5 , i.e., the time when i_1 was last scanned by the previous reader, as timeout information. Since i_1 moved to a new sensor, a new path p_8 is generated. Finally, we update `RFID_READ` for i_1 . When i_2 is scanned by s_3 , only the information for that item is updated to reference path p_8 . For all other incoming events at that time, only the cache has to be updated.

The resulting tables in the warehouse are depicted as Tables 6 and 7.

3.2.2 Bulk Data Staging

In contrast to the tuple-wise data staging algorithm where the warehouse is updated on a per-tuple basis, the bulk data staging algorithm updates the warehouse for several items that changed their location in a batch.

To allow for bulk data staging, the location change of an individual item is not immediately propagated to the warehouse. For this purpose, we store the time when the item was scanned for the last time by the previous reader. Only if this information is already set, i.e., the item moves to yet another sensor, data staging is enforced. In such a case the cache update for the item is delayed to avoid overwriting the old item state. The old item state is necessary for a correct data staging. After the data staging the cache is updated.

The first step of bulk data staging is to select the set of candidate events for data staging. This set contains the following items: Let t_{last} denote the time of the last data staging run and t_{now} the current point in time. The algorithm selects all entries from the cache table `RFID_RAW` whose values for column `timestamp` are in $[t_{last}, t_{now}[$. The selected events are all events that may have had a location change in the past. Since scanning individual items is not an atomic operation, we must exclude the timestamp t_{now} , i.e., probably not all events of t_{now} are already in the cache. Events with timestamp t_{now} are processed in the following data staging run.

This set constitutes the input for the bulk data staging algorithm shown in Figure 6. It is first joined with `RFID_READ` to get the current path p_{n-1} for each item in the set. The set contains one or more bulks – identified by their pioneer items – that must be extracted for

Input: Set of tuples $\{\langle \text{item} : e, \text{reader} : s_n, \text{t_in} : t_n^F, \text{timestamp} : t_n, \text{subpath_out} : t_{n-1}^L \rangle\}$, transferred from `RFID_RAW`, where e denotes the EPC of the scanned item, s_n the sensor that scanned e , t_n^F the time when e reached the range of s_n , t_n the timestamp, when s_n scanned e most recently, and t_{n-1}^L the time when the previous sensor last scanned e .

1. Left outer join input with `RFID_READ` on `item`. Resulting tuples $\{\langle e, s_n, t_n^F, t_n, t_{n-1}^L, p_{n-1} \rangle\}$, where p_{n-1} denotes the path id of the path the item e had traversed so far, if it exists.
 2. Group this resulting set by reader, path, incoming and current timestamp to obtain a set of bulks.
 3. For all bulks B of this set:
 - (a) Select the first item of the bulk as pioneer item e_p .
 - (b) Generate path information for pioneer item e_p only:

IF $(p_{n-1} = \text{null})$ THEN (*The item has been scanned for the first time*)

Create new path $p_1 : \langle p_1, \text{null}, s_1, "p_1", "s_1", t_1^F, \text{null} \rangle$ for e_p in `RFID_PATH`

ELSE (*The item is scanned by a new sensor.*)

 - i. Retrieve e 's path p_{n-1} , currently referenced in `RFID_READ`'s attribute `id`.
 - ii. **Terminate paths** (see (★) in Figure 5)
 - iii. Create path $\langle p_n, p_{n-1}^{\text{TERM}}, s_n, \text{id}_{s_n}, \text{sens}_{s_n}, t_n^F, \text{null} \rangle$ for e_p .
 - (c) Update `RFID_READ` for all items in e_p 's bulk, set `id` = p_n and `t_in` = t_n^F for e_p and all items e within the bulk.
-

Figure 6: Bulk Data Staging

further processing. The path information created with the pioneer items is then applied to all other items inside the bulk.

If the pioneer item was never scanned before, there are no matching entries in the tables `RFID_READ` and `RFID_PATH`. In this case a path is created. For a new pioneer event $\langle e, s_1, t_1^F, t_1, \text{null} \rangle$ the tuple $\langle p_1, \text{null}, s_1, "p_1", "s_1", t_1^F, \text{null} \rangle$ is created in `RFID_PATH`. Instead of creating new paths, no longer referenced paths can be reused.

If a sensor already scanned an item, there are existing entries in tables `RFID_READ` and `RFID_PATH`. As in tuple-wise data staging, the algorithm retrieves this information, and then determines whether the item has moved to a new sensor or it stayed at its old position.

If an item has moved, the currently stored path p_{n-1} must be terminated accordingly. The procedure is similar to the tuple-wise data staging. At last, a new path is created with the terminated path as predecessor and with `t_in` set to t_n^F . This new path is then referenced from the items inside the bulk.

The path information is only computed for the pioneer item of a bulk. The resulting path ID is cached, so it can be applied to all other events inside the bulk without a need for calculating their path information. A new calculation is only necessary if the algorithm identifies a new bulk inside the set of candidates.

Example This example is based on the same movements and scan times of the items as for the previous example. The path of the items can be found in Figure 4 and the scan times in Table 8. As before, the tables in the cache and in the warehouse are empty at the

beginning.

Times t_1 and t_2 : (Only cache operation, no data staging.)

Time t_3 : When item i_5 reaches the range of s_4 at t_3 , data staging is triggered. The set of data staging candidates contains all items with `timestamp < t3` stored in the cache. This selects only items i_5 and i_6 , because the cache has not been updated for these items. During data staging, a new path p_1 , which represents that the items were scanned by s_1 for the first time at t_1 , is generated. After that, the cache is updated for items i_5 and i_6 to reflect the movement of the items from s_1 to s_4 .

Time t_4 : When item i_1 reaches the range of sensor s_2 for the first time, data staging is triggered again. The set of candidates contains all items with $t_3 \leq \text{timestamp} < t_4$, i.e., items i_1 to i_4 . The items belong to two bulks, represented by the pioneer items i_1 (the bulk that has been at s_1) and i_5 (the items that moved to s_4). For i_1 , the path p_1 can be used because the `t_out` information is not set for this path.

Item i_5 is currently located on the not yet terminated path p_1 . Path p_1 is terminated with t_2 , and a new path p_2 with predecessor p_1 is created. For i_5 and i_6 `RFID_READ` is updated with the newly generated path.

Time t_5 : Data staging is triggered when item i_3 arrives at s_4 . The set of candidates contains all events with $t_4 \leq \text{timestamp} < t_5$, including pioneer items, i_3 and i_5 . Currently, item i_3 is on path p_1 , so a path similar to p_1 , but with termination time t_3 is needed. There is no path with these characteristics, so the algorithm duplicates p_1 and creates path p_3 with the correct `t_out` information. Path p_3 is then used as predecessor path for p_4 , which represents the movement of items i_3 and i_4 to s_4 . The path p_4 is linked to items i_3 and i_4 . Finally the pending cache update is done and the remaining events are processed.

Time t_6 : The event for item i_1 triggers data staging. The data staging candidates are all items with $t_5 \leq \text{timestamp} < t_6$. The set of candidates contains three bulks represented by three pioneer items. For the bulk that moves along with pioneer item i_1 , the algorithm finds the non-terminated path p_4 as a suitable path to be further used. Pioneer item i_3 finds the path p_4 , too, but terminates it with `timestamp t4`. Then the new path p_5 is created with p_4 as predecessor and linked to the bulk. At last, item i_6 takes the not terminated path p_2 , terminates it with t_4 and creates a new path p_6 . After this data staging no further data staging is needed, and only cache updates remain for the following incoming events.

For the last data staging run all events are selected, whose `timestamp` represents a time after the previous data staging run. If we apply this to our example, the last pioneer item is item i_1 . This item is currently on path p_4 . Searching for a suitable path, the algorithm does not find any, so p_4 must be duplicated. The resulting path p_7 is then used as predecessor path for path p_8 . The last path is then used for items i_1 and i_2 . All other bulks did not move, so no further processing action is necessary.

4 Evaluation

In this section, we will compare the different database schemas, described in Section 2.3, by their space requirements and their query performance. Furthermore, based on estima-

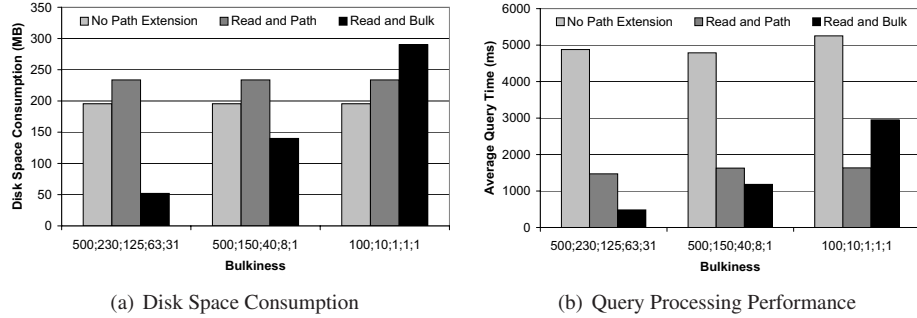


Figure 7: Comparison of Different Data Models

tions about the event rate of the German retailer Quelle and the car manufacturer BMW, we will assess the performance of the data staging algorithms.

4.1 Comparison of the Different Data Warehouse Schemas

As stated in Section 3.2, most items tend to move in bulks. For a comparison of the different database schemas, we adopted the tree model of [GHLK06]: Each node in the tree represents a location where a set of items has been scanned, while an edge represents the movement of objects between two locations. This model assumes that items at locations near the root move in larger groups, and with increasing tree depth the groups become smaller. The *bulkiness* $\mathcal{B} = (s_1; s_2; \dots; s_n)$ defines the maximum size s_i of a group at level i , with n being the *path length*. Our assumption implies $s_i \geq s_j$ for $i < j$. For example, a bulkiness of (10; 4; 2) denotes that a bulk at level 1 contains ten items, at level 2 there are two bulks with four items each, and one with two items, and at level 3 each previous bulk is split into bulks containing two items.

The data set used for this comparison consists of 5 million entries, generated by 1 million items passing by 5 RFID sensors. The query *What is the average time for product X to go through locations L_1, \dots, L_k entering L_k between times t_1 and t_2 ?* was chosen, since it has to walk through all these entries. In order to evaluate the impact of the size of the travel groups, we defined three different degrees of bulkiness $\mathcal{B}_1 = (500; 230; 125; 63; 31)$, $\mathcal{B}_2 = (500; 150; 40; 8; 1)$, and $\mathcal{B}_3 = (100; 10; 1; 1; 1)$.

As shown in Figures 7(a) and 7(b) the space requirements and the query processing time using the *No Path Extension* and the *Read and Path* data schema are independent of the bulkiness. These data schemas store the movement of an item on a per-item level. In contrast to that, the *Read and Path* schema stores the movement of items on a per-bulk level, resulting in less entries to be stored per item. Thus, the higher the size of the bulks the higher is the reduction of disk space usage with the *Read and Bulk* schema. Using schemas *No Path Extension* and the *Read and Path*, the path information for the example query must explicitly be reconstructed by n self-joins where n is the path length. With the *Read and Bulk* approach, this information can be easily obtained with a single join between tables `RFID_READ` and `RFID_PATH`. For large bulks, this results in a significant speed-up in query processing performance, as shown in Figure 7(b).

If the bulk sizes along the path for all items are small, the space requirements and the query processing time using the *Read and Bulk* schema are higher than using the other approaches. But in practice, we even expect bigger bulks than the ones we used for the presented scenario with 1 million items passing 5 sensors. Large containers or pallets can carry many more items. 5 RFID sensors constitute a rather low estimate considering that items are tracked throughout a supply chain. Thus, the *Read and Bulk* schema benefits even more compared to the other data schemas.

4.2 Estimation of Event Rates

We analyze the requirements of two representative companies: Quelle, a retailer and BMW, a manufacturer.

According to their website, the German mail order company Quelle ships 771,000 items per day from their most modern facility at Leipzig. Let us assume that each item passes by five RFID sensors, and that the facility is on duty eight hours a day. To satisfy these needs, the RFID architecture must handle at least 133 RFID events per second.

The values above model just the outward movement of goods. The event rate increases if automated stock keeping, where stock items are scanned frequently, is also modeled.

In 2005, BMW manufactured about 1.2 million cars, each consisting of about 20,000 parts. We assume that BMW tags about 1,000 of these 20,000 compounds with RFID tags, and 20 events per part are generated before the part is built into the car. Furthermore, we assume that BMW has 16 working hours a day and 220 working days per year. Together, all BMW facilities produce about 1,890 events per second. If we look at the facility at Leipzig, where currently 650 cars per day are produced, the event rate is about 226 events per second.

4.3 Evaluation of the RFID Data Staging Algorithms

The focus of the last part of this section is the performance analysis of the data staging algorithms.

Benchmark Configuration For the benchmarks, the middleware and the cache were running on the same host, equipped with an Intel Xeon 2.80 GHz CPU and 1 GB main memory. Since the cache and the middleware were placed on the same host, the JDBC driver used DMA for data transfer. The warehouse was placed on a host with a dual Intel Xeon 3.20 GHz EM64T CPU and 8 GB main memory. The simulated RFID sensors and the query clients were distributed over 8 hosts whose performance characteristics were similar to the middleware host. For every sensor or client, a dedicated socket connection to the middleware is opened.

Three different parameters – as explained in Figure 8 – affect the performance of the RFID architecture.

Since the number of items has no influence on the performance, as long as the dataset fits into main memory, this value is constantly held at 10,240. The path length has only an impact on the storage consumption inside the data warehouse, so this parameter was set to 5. The impact of storage consumption has been discussed above.

Bulkiness	The bulkiness specifies the hierarchical splittings of the individual bulks. This models a typical supply chain where items initially moved in larger containers.
Duplicate reads	This parameter specifies the number of scans at a certain location. At a given location, the event is at least scanned once, but not more often than the value of this parameter. The intention behind this parameter can be found in stock keeping. If an item stays at a certain location for a longer period of time, an RFID reader may scan the item more than once.
OLTP/OLAP query ratio	Every query client issues OLTP or OLAP queries. The ratio between these two types is specified by this parameter.

Figure 8: Benchmark Parameters

The evaluation system creates a tree of bulk movements, similar to the one used previously in the examples. To model some fuzziness, some of the leaves are cut off randomly, such that less than 1% of the items get lost.

RFID Queries The evaluation system generates some background load using different lightweight OLTP queries on the cache data and heavy OLAP queries executed on the historic data in the warehouse. The ratio between these types of queries is defined as described in Figure 8. An example lightweight query is

Where is item i now?

whereas a heavy OLAP query is

What items need more time to get from l_i to l_j than the average of items of the same type?

We implemented several different query types for OLTP and OLAP but omit the description due to space limitations.

Results For each of the parameters in Figure 8, we performed different benchmarks. To evaluate the impact of a parameter on the throughput for data staging, we always varied one parameter while holding the remaining two parameters constant.

The basis for all benchmarks are 10,240 items which pass 5 RFID sensors with a bulkiness of $\mathcal{B}_2 = (10,240; 3,520; 320; 80; 5)$. Every item is scanned twice by the same reader before moving to another location. 30% of the user queries are OLAP queries. This standard case is represented by the middle bars in Figures 9(a), 9(b), and 9(c).

Figure 9(a) shows the impact of the bulkiness on the performance of the data staging. The parameter *bulkiness* varies the number of items that travel together in one group. Bulkiness $\mathcal{B}_1 = (10,240; 860; 75; 30; 5)$ denotes a scenario where the groups of items traveling together are smaller than in the standard case. In contrast to that, bulkiness $\mathcal{B}_3 = (10,240; 6,500; 3,000; 1,250; 320)$ models bigger groups of items along the path. While the throughput with tuple-wise data staging is nearly constant (variation of less than 7%), the impact of bulk data staging is quite high. There is an 25% performance improvement with big bulks and less splits because a higher number of items are processed in one batch in bulk data staging.

The number of *duplicates*, i.e., how often the same sensor scans an item, has a significant impact on the performance of data staging, as shown in Figure 9(b). While the throughput

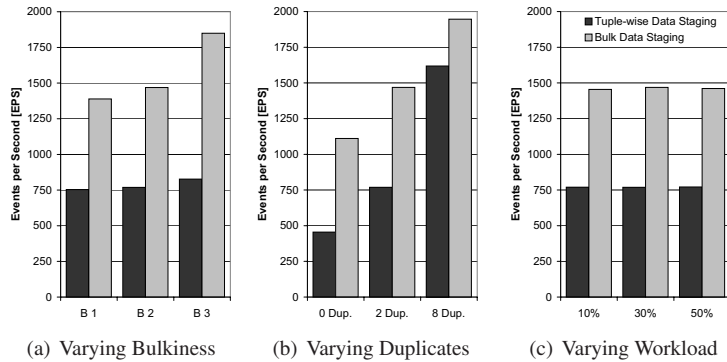


Figure 9: Benchmark Results

for tuple-wise data staging is doubled if all events are scanned eight times instead of two times, the bulk data staging has a performance boost of one third. The more duplicate reads there are, the more transactions can be handled entirely by the cache, i.e., there are no updates of the warehouse.

As depicted in Figure 9(c), the variation of the *ratio* between lightweight (OLTP) and heavyweight (OLAP) queries has no impact on the throughput of event processing. This shows that the limiting factor for the performance of the data staging is not the warehouse, but the cache and the middleware. We use this as an indicator that our architecture has a scalability potential by replicating and distributing the main-memory caches. Handling RFID events and queries with distributed caches is ongoing work and is beyond the focus of this paper.

The average event processing throughput of our system is about 1500 events per second for bulk data staging, and about 850 events per second for tuple-wise data staging. As expected, bulk data staging performs better for two reasons: First, events are processed in efficient batch operations within the warehouse and second, the data staging process is triggered less often.

The data staging algorithms does not have significant impact on the query execution time of lightweight and heavyweight queries. While the data staging algorithms are running, there was no noteworthy performance degradation (between 1% and 3%) for the individual queries, compared to the data stores not loaded with any data staging algorithm.

5 Related Work

Many application service vendors, including IBM [IBM], HP [HP], Microsoft [Mic05], Oracle [Ora], SAP [BLHS04], Siemens [WL05, WLLB06], and Sun [GS04] are working on their own RFID middleware infrastructures. These frameworks transform the proprietary events stemming from the RFID sensors into events that can be processed by high-level applications. The generic design of such RFID middleware solutions does not explicitly store paths of tracked items, although RFID events are automatically stored in the database. Our approach could thus be viewed as an extension which exists on top of existing RFID middleware.

There are different approaches to reduce the storage space for RFID data in databases. Hu et al. [HSCS05] describe an Oracle bitmap datatype which exploits the structure of the EPC to reduce the amount of data to be stored. The bitmap datatype represents a collection of EPCs which share a common prefix, e.g., items stemming from the same manufacturer. For all items, the prefix is only stored once. To further reduce the space requirements, a compressed representation of the EPC suffixes (the serial numbers of the items) is maintained in the data type. Wang and Liu [WL05] propose a temporal data model for RFID data for object tracking and monitoring. In order to reduce the storage space for the RFID data, they aggregate information by exploiting (hierarchical) containment relationships among objects, e.g., a pallet loaded with cases. In contrast to our approach, the data model does not store the path of a tracked object.

An approach similar to our work is presented by Gonzales et al. [GHLK06]. They use RFID Cuboids to store aggregated path information at different abstraction levels. Although they identify incremental updates as crucial point for RFID applications, their work does not describe incremental updates of the path information in the warehouse.

6 Conclusion and Future Work

In this work we devised an information system for supporting OLTP and OLAP queries on RFID data at the same time. It proposes algorithms for generating the path information at data staging time, to facilitate OLAP queries for RFID tracking and supply chain analysis. Different data warehouse schemas for storing the path information are evaluated and implemented in an RFID tracking infrastructure, consisting of a main memory DBMS used as cache and an RDBMS used as a data warehouse.

Our evaluation shows the impact of different factors on the data staging algorithms. Bulk data staging leverages the efficient bulk operations of DBMSs and demonstrates its advantage over propagating events to the data warehouse tuple by tuple.

Supporting multiple caches is part of our ongoing work. We are implementing an RFID routing system that passes the incoming events to distributed caches, whereby a specific item is always in the same cache. Further development consists of dynamic routing and load balancing strategies. In the future, we will investigate how much of the routing functionality can be placed inside intelligent networks, e.g., Cisco SONA, or inside a publish-subscribe-architecture, where the RFID sensors publish their data streams and the caches subscribe those events that are interesting for their clients. In this context we will also investigate RFID application-specific data stream mining techniques, that, e.g., raise alarms, if certain supply chain abnormalities (correlated events) occur.

Acknowledgments

We would like to thank Wolfgang Becker, Tobias Brandl, and Dr. Ulrich Marquard of SAP's Research and Breakthrough Innovation Technology and Foundation group for their cooperation. Also, we would like to thank Holger Pawlita for helping in the implementation of the Evaluation System, and Christian Sosnowski for the detailed comparison of the database schemas.

References

- [BLHS04] Christof Bornhövd, Tao Lin, Stephan Haller and Joachim Schaper. Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1182–1188, 2004.
- [CKRS04] Sudarshan S. Chawathe, Venkat Krishnamurthy, Sridhar Ramachandran and Sanjay Sarma. Managing RFID Data. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1189–1195, London, UK, 2004. Springer.
- [EPC06] EPCglobal: EPC Tag Data Standards Version 1.3, Ratified Specification. <http://www.epcglobalinc.org/standards/>, March 2006.
- [GHLK06] Hector Gonzalez, Jiawei Han, Xiaolei Li and Diego Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 83–94, 2006.
- [GS04] Alka Gupta and Mayank Srivastava. Developing Auto-ID Solutions Using Sun Java System RFID Software. java.sun.com/developer/technicalArticles/Ecommerce/rfid/sjsrfid/RFID.html, October 2004.
- [Hei05] Claus Heinrich. *RFID and Beyond – Growing Your Business Through Real World Awareness*. Wiley Publishing, 2005.
- [HP] HP. HP OpenView for the RFID Environment: Supporting an Uninterrupted Supply Chain. h20229.www2.hp.com/solutions/mfg/sg/mfg_sg_rfid.bb.pdf.
- [HSCS05] Ying Hu, Seema Sundara, Timothy Chorma and Jagannathan Srinivasan. Supporting RFID-Based Item Tracking Applications in Oracle DBMS Using a Bitmap Datatype. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1140–1151, 2005.
- [IBM] IBM. WebSphere RFID Premises Server. http://www-306.ibm.com/software/pervasive/ws_rfid_premises_server/.
- [JGF06] Shawn R. Jeffrey, Minos Garofalakis and Michael J. Franklin. Adaptive Cleaning for RFID Data Streams. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 163–174, 2006.
- [Mic05] Microsoft Unwraps Technology at TechEd 2005 to Spark Greater Adoption of RFID. <http://www.microsoft.com/presspass/press/2005/jun05/06-07RFIDTechEdPR.mspx>, June 2005.
- [Ora] Oracle. Sensor Edge Server. http://www.oracle.com/technology/products/sensor_edge_server/index.html.
- [OTS⁺06] Karn Opasjumruskit, Thaweesak Thanthipwan, Ohmmarin Sathusen, Pairote Sirinamattana, Prachanart Gadmanee, Eakkaphob Pootarapan, Naiyavud Wongkomet, Apinunt Thanachayanont and Manop Thamsirianunt. Self-Powered Wireless Temperature Sensors Exploit RFID Technology. *IEEE Pervasive Computing*, 5(1):54–61, 2006.
- [TDF06] Frédéric Thiesse, Markus Dierkes and Elgar Fleisch. LotTrack: RFID-Based Process Control in the Semiconductor Industry. *IEEE Pervasive Computing*, 5(1):47–53, 2006.
- [WL05] Fusheng Wang and Peiya Liu. Temporal Management of RFID Data. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1128–1139, 2005.
- [WLLB06] Fusheng Wang, Shaorong Liu, Peiya Liu and Yijian Bai. Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 588–607. Springer, 2006.