# A Deferred Cleansing Method for RFID Data Analytics

Jun Rao Sangeeta Doraiswamy Hetal Thakkar[1] Latha S. Colby
{junrao,dsang}@us.ibm.com hthakkar@cs.ucla.edu lathac@us.ibm.com
IBM Almaden Research Center UCLA IBM Almaden Research Center

## ABSTRACT

Radio Frequency Identification is gaining broader adoption in many areas. One of the challenges in implementing an RFID-based system is dealing with anomalies in RFID reads. A small number of anomalies can translate into large errors in analytical results. Conventional "eager" approaches cleanse all data upfront and then apply queries on cleaned data. However, this approach is not feasible when several applications define anomalies and corrections on the same data set differently and not all anomalies can be defined beforehand. This necessitates anomaly handling at query time. We introduce a deferred approach for detecting and correcting RFID data anomalies. Each application specifies the detection and the correction of relevant anomalies using declarative sequence-based rules. An application query is then automatically rewritten based on the cleansing rules that the application has specified, to provide answers over cleaned data. We show that a naive approach to deferred cleansing that applies rules without leveraging query information can be prohibitive. We develop two novel rewrite methods, both of which reduce the amount of data to be cleaned, by exploiting predicates in application queries while guaranteeing correct answers. We leverage standardized SQL/OLAP functionality to implement rules specified in a declarative sequence-based language. This allows efficient evaluation of cleansing rules using existing query processing capabilities of a DBMS. Our experimental results show that deferred cleansing is affordable for typical analytic queries over RFID data.

## 1. Introduction

Radio Frequency Identification (RFID) technology is being deployed in several application areas including supply-chain optimization, business process automation, asset tracking, and problem traceability applications. While RFID itself is not a new concept, Electronic Product Code (EPC™) [1] standards-based product identification and tracking is emerging as a key component in the enablement of these applications. EPC is a scheme for uniquely identifying individual objects using RFID tags and other means. One of the challenges in implementing an RFID-based system is dealing with anomalies in the data representing RFID reads. These data can have errors arising from many different sources.

Unlike barcode readers that detect signals based on a fixed line of sight, RFID readers need to respond to signals from a wider range of objects that could potentially be moving. The complexities of "reader physics" give rise to duplicate reads, missed reads, and cross reads. In addition to errors arising from physical reads, anomalies can occur at the logical or business process level. For example, suppose that a product item moves to a store floor but is returned to the back-room due to lack of shelf space and at a later time is again moved to the store floor; and this cycle repeats a few times. An application may treat such a cycle as an anomaly and want to remove it as erroneous data, even though the raw reads are themselves accurate.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
*VLDB '06*, September 12–15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

The conventional data cleansing approach is to remove all anomalies upfront and to store only the cleaned data in a database. For example, many device controllers at the edge of an RFID network provide de-duping and primitive filtering capabilities and errors such as duplicate reads are often correctible at the edge. Such eager cleansing methods can potentially reduce the amount of data that have to be managed by applications down-stream in the business process and help avoid repeated cleansing of the data at query time.

However, it is not always possible to remove all anomalies before hand. One reason is that the rules and the business context required for cleansing may not be available at data loading time. For example, we may not know the presence of cycles and whether they will affect any analysis until users observe irregularity in query results some time later. As a result, an application may constantly evolve existing anomaly definitions and add new ones. Also, the rules for correcting data anomalies are often application specific. For example, application queries tracking shelf space planning or labor productivity will want to know about all cycles within stores. On the other hand, another application that calculates how long a product item has stayed in every location will want to remove everything in the cycle except for the first and the last reads. Finally, in applications such as pharmaceutical e-pedigree tracking, laws require the preservation of tracking information which then precludes up-front cleansing of the data. Maintaining and adapting multiple cleaned versions physically is prohibitive, when different application requirements dictate sets of rules that are dynamically changing. All these reasons make the eager approach infeasible.

We propose a deferred cleansing approach to complement the eager one. Known anomalies whose detection and correction is common to all consumers of the data are still handled eagerly, but the processing of other anomalies is deferred until query time. Each application specifies its own anomalies by defining cleansing rules. The rules do not change the content of the database directly, but are evaluated when an application issues a query. In this approach, although an application pays some cleansing overhead at query time, it gains the flexibility of being able to evolve its anomaly specifications over time.

[1]Work performed at IBM Almaden.

A key characteristic of RFID data is its sequential nature. We first describe different types of anomalies that can occur in RFID data and show the patterns in the read sequences that are indicative of those anomalies. We define cleansing rules in an extended version of SQL-TS[3]—a simple yet powerful sequence-based language, and implement each rule in SQL/OLAP for efficient evaluation. The naïve way of performing deferred cleansing is to clean all the data on the fly before executing a query. This is prohibitive since RFID data has high volume. We develop two query rewrite techniques that effectively exploit query conditions to reduce the amount of data to be cleaned at query time. We have built a prototype implementation of the deferred cleansing model. Our experimental results show that deferred cleansing is affordable for typical RFID analytical queries and our rewrite techniques improve query performance significantly compared to the naïve evaluation.

The rest of this paper is organized as follows. Related work is described in Section 2. An overview of our deferred cleansing system is provided in Section 3. Section 4 describes our approach for specifying cleansing rules and the exploitation of SQL/OLAP. It also provides example rules for detecting and correcting the errors in several scenarios. In Section 5, we present mechanisms for rewriting user queries based on cleansing rules and show how these rewrites can be optimized. We explain experimental results that evaluate the benefits of deferred cleansing in Section 6 and present our conclusions in Section 7.

## 2. Related Work

There are several challenges in dealing with RFID data [2]. Techniques for compactly representing RFID data along with methods for accessing the data have been proposed by [16] and [17]. In this paper, we focus on the challenge of dealing with anomalies in the data. A data warehousing solution often contains data cleansing as one of the steps in the ETL (Extract-Transform-Load) process before the data is loaded into the data warehouse. Commercial ETL products such as, [12] and [13] provide data cleansing and profiling capabilities. These techniques involve detecting and correcting 1) duplicate representations of the same entity (e.g., customer), 2) references to missing data – based on referential integrity constraints, and 3) data that are inconsistent with some standard reference data (such as names and addresses information obtained from postal directories). There has been some recent research ([14] and [15]) in applying fuzzy operators to the correction of these types of anomalies.

Data generated from RFID reads adds another dimension of complexity to the cleansing process. Cleansing RFID data requires analyzing product lifecycle information which involves detecting and correcting errors across sequences of facts. Duplicate reads (caused by the same reader reading a tag continuously) are often easily corrected at the edge of the EPC network and edge devices manufactured by several companies provide this form of de-duping capability. Recently, companies like OAT systems [11] have also started to offer more complex error detection and correction capabilities in the edge systems. SAP's Auto-ID infrastructure described in [6], provides data filtering, enriching, and aggregation components in the device control layer. Our solution, on the other hand, can compensate for errors that persist beyond the edge and deal with application specific data quality requirements.

In [4], the authors describe a system for transforming low-level device-specific and error-prone data into idealized data that can be processed by higher-level applications. A declarative language is used to define cleansing operations that involve de-duping, removing outliers (using standard deviation), and smoothing data collected from different sources. Wang and Liu [5] describe a system for managing RFID data based on an extended ER model. They provide example rules for data filtering (deleting duplicate reads by the same reader), and inferring aggregation events (e.g., when pallets have been loaded on to a truck). Both of these approaches cleanse data up-front using a fixed set of rules. Our approach, on the other hand, allows application specific cleansing at query time.

There has been a large body of work on consistent query answering over inconsistent databases (e.g., [9]). These papers study the problem of answering queries posed over an inconsistent database to generate answers over a consistent version of the data as defined by consistency constraints. Some of them focus on the complexity of query answering in the presence of inconsistencies. Others ([7][8][10]) have presented techniques for efficiently rewriting the queries to provide consistent answers, i.e., queries whose result sets represent the result of applying a possible "repair". The class of constraints studied is limited to those requiring set (or bag) based relational semantics. Our focus is on a different but related problem. In this paper, we assume that the consistency rules specified by the user include not only the consistency definition but also the action that specifies the "repairs". The class of applications that we have focused on requires analysis of sequences to detect and correct anomalies. A naïve approach to evaluating these sequence-based consistency rules would be extremely inefficient. We present techniques for combining these rules with user queries to generate rewritten queries that are not only efficient to execute but also preserve the semantics of the sequence constraints.

In [3], Sadri et al, propose a pattern language, namely SQL-TS, and optimization techniques for its efficient evaluation. These optimization techniques involve capturing the logical relationships between the pattern elements based on the specified predicates. The approach proposed in this paper also involves analyzing the predicates of the SQL-TS rules. However the two techniques are very different in many respects. For instance, [3] determines dependencies among the predicates of the SQL-TS query itself to prevent redundant pattern matching, whereas our technique compares SQL-TS rule predicates with the predicates specified in the user query to determine the data that needs to be cleaned. The analysis of the predicates is performed very differently and for very different reasons.

In [18], Deshpande et al, describe techniques for querying live data from sensor networks by incorporating statistical models. These models can provide approximations and extrapolations of missing and faulty readings. Queries are answered by calculating estimates of current readings from the constructed model. Finally, view adaptation [20] techniques can be applied to maintain materialized results when the view definition evolves. However, such techniques are not directly applicable on sequence data.

## 3. System Overview

Our implementation of the deferred cleansing approach comprises two main components - the Cleansing Rule engine and the Query

Rewrite engine. Both components are prototyped in Java above the DBMS, are extensible, and are not proprietary to any DBMS vendor. A high level architecture diagram is depicted in Figure 1.

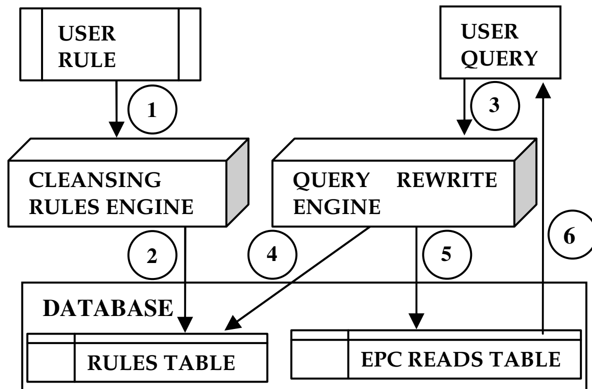


**Figure 1: System Architecture**

The numbered steps in Figure 1 represent the flow of information in the system and are summarized below.

1. The rule engine accepts rules specified in extended SQL-TS for different applications and generates a SQL/OLAP template for each rule. This SQL/OLAP template encapsulates logic to detect and compensate for anomalies and is plugged in at query time.
2. The SQL/OLAP template is persisted in the rules table. Specifically, the rule pattern, conditions, and action clauses are stored in this table for use by the rewrite engine.
3. The rewrite engine intercepts user SQL queries to determine if they need to be re-written to compensate for errors.
4. If needed, the user query is rewritten to a new one by efficiently applying relevant rules stored in the rules table.
5. The rewritten query is submitted to the DBMS for execution.
6. The DBMS returns cleansed query results to the user.

# 4. Cleansing Rules

In this section, we describe how we detect and correct anomalies using declarative cleansing rules. We assume that the reads of all RFID tags are stored in a relational table R whose schema is depicted in Figure 2. This schema is used throughout the paper.

| Table R (for RFID reads) | |
|---|---|
| epc, | RFID tag identifier |
| rtime, | time when tag is read by a reader |
| reader, | identifier of reader that reads the tag |
| biz_loc, | business location where tag is read |
| biz_step, | business steps associated with the read |

**Figure 2: RFID Table Schema**

## 4.1 Cleansing Rules Language Considerations

We can naturally view RFID data as a set of EPC sequences, each of which consists of all reads of a particular tag in *rtime* order. Such a model makes it convenient to detect various types of anomalies for RFID applications. Many sequence-based languages have been proposed in the literature. However, few have recognized that SQL/OLAP functions (part of the SQL99 standard) can be exploited for processing sequences. For example, to detect and filter duplicate reads, we can use this SQL statement:

```
with v1 as (
  select biz_loc as loc_current,
      max(biz_loc) over (partition by epc  order by rtime asc
      rows between 1 preceding and 1 preceding) as loc_before
  from R    )
select * from v1
where loc_current != loc_before or loc_before is null;
```

The key in the above statement is the utilization of SQL/OLAP for computing column loc_before. SQL/OLAP uses the partition by and the order by clauses to define the input as a set of sequences, and allows us to define for each row r, a scalar aggregate on a window (within a sequence) relative to r. In this example, we define the input as EPC sequences and specify a window (given by the rows clause) with a single row before each row r in sequence order. The biz_loc of the previous row in the window can be extracted using a scalar aggregate. We can then compare the location of two consecutive reads to remove duplicates. The second disjunct in the where clause handles border line rows that have no rows before them in a sequence.

Exploiting SQL/OLAP for sequence processing has several advantages: (1) it is more efficient than SQL using self joins or subqueries (the above SQL statement can be executed by making a single pass over table R in a sorted order); (2) it is integrated inside the database engine and therefore automatically benefits from DBMS features such as optimization and parallelism; (3) it is standardized and supported by the leading DBMS vendors.

The main drawback of using SQL/OLAP is its redundancy in syntax. Observe that in order to retrieve a column value from a previous row, one has to specify a relatively complex scalar aggregate. If multiple columns are needed, each requires its own scalar aggregate specification. Such redundancy makes it hard for users to express cleansing rules directly in SQL/OLAP.

To overcome such drawback, we extend the syntax of SQL-TS, a natural sequence language, to make it easier to define cleansing rules. Once a rule is defined, we automatically generate a template in SQL/OLAP to be used at query time for efficient execution. Section 4.2 introduces a SQL-TS based rule language and shows how rules expressed in such a language can be mapped to a SQL implementation. We then use several examples to demonstrate how our language can be used to specify cleansing rules in Section 4.3. We discuss the issue of rule ordering in Section 4.4.

## 4.2 SQL-TS based Cleansing Rules

Our SQL-TS based rule grammar is described below; all clauses except for those in *italics* are borrowed as-is from SQL-TS.

```
DEFINE        [rule name]
ON            [table name]
FROM          [table name]
CLUSTER BY    [cluster key]
SEQUENCE BY   [sequence key]
AS            [pattern]
WHERE         [condition]
ACTION        [DELETE | MODIFY | KEEP]
```

The CLUSTER BY and the SEQUENCE BY clauses are similar to the ‘partition by’ and ‘order by’ clauses in SQL/OLAP and define how to convert the input data to sequence sets. Typically

the cluster key is *epc* and the sequence key is *rtime*. The main simplification comes from the pattern specification in the AS clause. A pattern defines an ordered list of references. If a reference is not designated with a * sign, it refers to a single row in the input. A reference with a * sign can only appear at the beginning or the end of the pattern, and refers to a set of rows either before or after a row bound to a singleton reference within a sequence. A condition is specified on columns of the pattern references. For example, duplicate detection can be expressed in SQL-TS as:

AS (A, B) WHERE A.biz_loc !=B.biz_loc

References A and B each refers to a single row and the pattern implies that the two rows are adjacent in a sequence. Compared to SQL/OLAP, the specification of the two consecutive reads in SQL-TS is simpler and more intuitive. A condition on a set reference (with a * sign) has the existential semantics, i.e., it is true if any row in the set makes it true (note that the semantics here is slightly different from the one used in the original SQL-TS). We show such an example in Section 4.3.

Our rule language extends SQL-TS in two ways. First, we add an ACTION clause that specifies how to fix anomalies when the condition in the WHERE clause is satisfied. An action is specified on a singleton reference defined in the pattern. We allow the action to be any of DELETE, MODIFY and KEEP. Using both DELETE and KEEP provides more flexibility since sometimes it is more intuitive for users to specify reads to remove, instead of retain, or vice versa. MODIFY can change the value of any column in a row. If a column to be modified does not exist, we create a new column on the fly. Note that an action does not directly change what is stored in the input table, but controls what flows out of it. We deliberately exclude INSERT from the ACTION clause, since often direct insertion is not useful. Second, we separate the table on which a rule is defined (by the ON clause) and the table from which the rule gets the input (by the FROM clause). We assume in the rest of the paper that a rule is always defined on the reads table R. However, an application can choose to use an input including data in R as well as some extra data for reference or compensation (e.g., handling missed reads in Example 5 in the next section). This is essentially how we support insertion without an explicit one in the ACTION clause. The input table is required to have a schema including all columns in R, but can have some extra ones. The rule condition can refer to any column in the input table.

We can always convert a cleansing rule to an implementation in SQL/OLAP. The conversion of the cluster by and the sequence by clauses is trivial and we focus on the conversion of the WHERE and the ACTION clauses. If a rule condition refers to two singletons, we can specify one of them in SQL/OLAP as a number of scalar aggregates (one for each needed column) over a window of size one. The window is defined according to the relative sequence positions of the two singletons. If a rule condition refers to a set and a singleton, we define a window including the set and convert the rule condition to a "case" expression over the window. A scalar aggregate is used to determine whether any row in the set tests to true. The DELETE and KEEP actions are implemented as filter conditions in SQL. The rule condition is used directly as the filter for KEEP and is negated for DELETE with proper handling of the null semantics. Finally, MODIFY can be handled by another "case" expression as seen in Section 4.3.

### 4.3 Cleansing Rule Examples

In this section, we use some examples to illustrate how cleansing rules are defined using the extended SQL-TS. We show only the pattern, condition and action specifications for each rule. Unless otherwise mentioned, a rule is always on table R, from table R, clustered by *epc*, and sequenced by *rtime*. We also highlight the interesting parts of the SQL/OLAP implementation of these rules. In this paper, we assume that the rules are manually specified. We leave the automatic discovery of such rules for future work.

**Example 1 (duplicate rule):** Although most duplicates can be fixed at the edge, a small number of them may survive for reasons such as edge server restart. We can further restrict duplicate removal to only reads that are t1 minutes apart, where t1 can be customized for different applications. Such a rule is given below. Here, we choose to keep the very first read among duplicates.

| Pattern | Condition | Action |
|---|---|---|
| (A, B) | A.biz_loc = B.biz_loc and<br>B.rtime – A.rtime < t1 mins | DELETE B |

The SQL/OLAP implementation is similar to that in Section 4.1.

**Example 2 (reader rule):** Consider a scenario where a forklift equipped with an RFID reader (say readerX) carries a tagged case to a destination in a warehouse. On reaching the destination, readerX reads the EPC on the case and a pre-installed location tag at the destination and generates a new read. During transportation, the case on the forklift may be accidentally read by other readers (say a reader on a docking door). If we discover that such transportation takes up to t2 minutes, we can define the following rule to remove all reads recorded t2 minutes before a read by readerX. Notice that B is designated as a set reference by the *.

| Pattern | Condition | Action |
|---|---|---|
| (A, *B) | B.reader = 'readerX' and<br>B.rtime – A.rtime < t2 mins | DELETE A |

The reader rule can be implemented in SQL/OLAP by defining a scalar aggregate "has_readerX_after":

```
max(case when reader = 'readerX' then 1 else 0 end)
    over (range between 1 macro sec following
        and  t2 min following) as has_readerX_after
```

Notice how we construct the window by exploiting the constraint on the sequence key rtime to include rows that B refers to. We can then filter out rows whose has_readerX_after is set to 1.

**Example 3 (replacing rule):** Suppose readers at two locations 'loc1' and 'loc2' are close to each other and can incur cross reads. Also suppose that because of the business flow, a shipment being read at 'loc1' is always read at another location 'locA' next, within t3 minutes. We can then use the following rule to detect the anomaly and modify its location.

| Pattern | Condition | Action |
|---|---|---|
| (A, B) | A.biz_loc = 'loc2' and<br>B.biz_loc = 'locA' and<br>B.rtime–A.rtime < t3 mins | MODIFY<br>A.biz_loc='loc1' |

The SQL implementation of the MODIFY action is a "case" expression that either keeps biz_loc as it is or changes it to 'loc1' depending on the test of a condition.

**Example 4 (cycle rule):** Suppose that an application does not want to see reads back and forth between a set of locations. Hence a location pattern such as [X Y X Y X Y] for an EPC should be changed to [X Y], keeping only the first X and the last Y. This can be achieved by the following cycle rule. A rule that removes cycles of arbitrary length is also possible, but more involved.

| Pattern | Condition | Action |
|---|---|---|
| (A, B, C) | A.biz_loc=C. biz_loc and<br>A.biz_loc != B.biz_loc | DELETE B |

**Example 5 (missing rule):** Let us take a scenario where a pallet and cases in it are known to travel together along a certain business path. Also assume that pallet tags are always readable, but due to the orientation of the tag and contents, cases are not always read at every location. Suppose that at location L1, pallet P is read, but a case C in P is not. We will be more confident that C in fact missed a read (instead of being stolen) if some time later, we see a read of C and P together again at some location L2. We can then compensate for the missing read of C at L1, by converting the pallet read at L1 to a case read (replacing P's epc with C's epc).

Below, we specify the missing rule in two sub-rules r1 and r2. Both r1 and r2 are defined on table R, which we assume for now, contains only case reads. The input (in the FROM clause) to r1 is not R, but a derived table of the same schema in Figure 2, with an extra column "is_pallet". We defer details on how this input is derived until Section 6. For now, let us assume that the derived table is a union of R and another set R'. For a pallet P containing n cases, there are n copies of every read of P in R' and the epc of each copy is set to each case epc. While R has the 'actual' reads for the cases, R' contains the 'expected' case reads based on the more reliable pallet reads. Every row in R and R' has is_pallet set to 0 and 1, respectively.

Rule r1 uses A to reference a pallet read and then checks if A has a nearby case read at the same location. If so, it sets a flag "has_case_nearby", which indicates no missing reads at that location. The output of r1 is pipelined to Rule r2. r2 keeps all original case reads, plus the pallet reads without a nearby case read, as long as the same case is read later together with the pallet. Observe that the preserved pallet reads compensate for the missing case reads.

| r1. Pattern | Condition | Action |
|---|---|---|
| (X,A,Y) | A.is_pallet=1 and<br>( (X.is_pallet=0 and<br>A.biz_loc=X.biz_loc and<br>A.rtime-X.rtime<5 mins)<br>OR<br>(Y.is_pallet=0 and<br>A.biz_loc=Y.biz_loc and<br>Y.rtime-A.rtime<5 mins)<br>) | MODIFY<br>A.has_case_nearby=1 |
| **r2. Pattern** | **Condition** | **Action** |
| (A,*B) | A.is_pallet=0 or<br>(A.has_case_nearby=0 and<br>B.has_case_nearby=1) | KEEP A |

To express r1 in SQL, has_case_nearby can be computed by a single scalar aggregate over a window including a row r, a row before and a row after r, since the condition on X and Y are the same. r2 can be dealt with methods similar to those in Example 2.

SQL/OLAP is richer than SQL-TS for expressing conditions. For example, if we change the scalar aggregate for computing "has_readerX_after" in Example 2, from max() to count(), we can further control how many reads by readerX should be observed before taking an action. Extending SQL-TS to take advantage of such capabilities in SQL/OLAP is beyond the scope of this paper since the current SQL-TS language is powerful enough to express many common types of anomalies.

### 4.4 Rule Ordering

When an application defines multiple rules on the same table, we require that their input table be the same. Often, there is a dependency among rules and their ordering is important. Consider the location of a sequence of tag reads given by [X Y X]. If we apply the cycle rule first, followed by the duplicate rule (without constraint on rtime), the cleaned sequence becomes [X] (first X). If we switch the two rules, we get [X X] instead. In our system, rules are ordered by their creation time and applied in this order.

## 5. Rewriting Queries using Cleansing Rules

Given a user query Q and a cleansing rule C defined on R, we denote the correct answer to Q with respect to C as Q[C]. We define $\Phi_C(d)$ as the result of applying rule C on a data set d including all columns in R's schema. By definition, Q[C] can be computed by replacing all references to R in Q with $\Phi_C(R)$. Such a computation requires cleaning all data in R and thus is prohibitive. In this section, we describe how deferred cleansing can be performed through more efficient query rewrites. In Section 5.1, we show why pushing predicates in Q to R directly does not always produce the correct answer and illustrate two efficient query rewrite approaches that preserve the query semantics. We describe techniques to generate these two types of rewrites in Section 5.2 and 5.3 respectively, in the presence of a single cleansing rule. We then extend our solution to support multiple rules in Section 5.4. In this section, we assume that the input to C is also table R, but our techniques apply to any input to C.

### 5.1 Motivation

To reduce the amount of data to be cleaned, it is tempting to push predicates in Q directly to R first and then to apply rule C, followed by the evaluation of the rest of Q. Unfortunately, this does not always return the correct answer for Q[C].

Consider the cleansing rule C1 (defined as the reader rule in Section 4.3) defined on table R1 and query Q1 given in Figure 3(a). We use an rid field to identify rows in a table. Applying C1 on R1 will remove row r1 because there is a read by readerX subsequently within 5 minutes. The remaining row r2 does not satisfy the condition in Q1 and the correct answer to Q1[C1] is {}. If we push Q1's condition "rtime < t1" on R1 first, only row r1 qualifies. Applying C1 on {r1} does not remove r1 this time since r2 is no longer present. Thus, we get the answer {r1}, which is incorrect.

As another example, consider a cleansing rule C2 defined on table R2 and another query Q2 given in Figure 3(b). Note that C2 is a modified version of the duplicate rule in Section 4.3 obtained by omitting the time constraint. Applying C2 on R2 produces {r3}

| C1: Pattern | Condition | Action |
|---|---|---|
| (A, *B) | B.reader = 'readerX' && B.rtime – A.rtime < 5 min | DELETE A |
| R1 (rid, epc, rtime, reader) = { ( r1, e1, t1-2min, 'readerY'), ( r2, e1, t1+2min, 'readerX') } | Q1: select * from R1 where rtime < t1 | |

(a) cleansing rule C1 defined on R1, queried by Q1

| C2: Pattern | Condition | Action |
|---|---|---|
| (E, F) | E.biz_loc = F.biz_loc | DELETE F |
| R2 (rid, epc, rtime, biz_loc) = { (r3, e2, t2-2 min, 'locZ'), (r4, e2, t2+2 min, 'locZ') } | Q2: select * from R2 where rtime > t2 | |

(b) cleansing rule C2 defined on R2, queried by Q2

| | |
|---|---|
| C1 | cr1: A.rtime<B.rtime, A.epc=B.epc, B.rtime<A.rtime+5min, B.reader='readerX' |
| Q1 | s1 : A.rtime < t1<br>cc1: B.rtime <t1+5min && B.reader='readerX'<br>ec1: rtime<t1+5min && (rtime<t1 \|\| reader='readerX') |

(c) deriving expanded rewrite for Q1[C1]

| | |
|---|---|
| C2 | cr2: E.rtime<F.rtime, E.epc=F.epc, |
| Q2 | s2 : F.rtime > t2<br>cc2: { } |

(d) deriving expanded rewrite for Q2[C2]

**Figure 3**. Running Examples

because r4 is a duplicate. Since r3 has an rtime no greater than t2, the correct answer for Q2[C2] is again {}. However, if we apply the condition in Q2 on R2 first before cleansing, only r4 is selected. Applying C2 on {r4} does not remove r4 since it is the only one in the set. Again, we get the incorrect answer {r4}.

Both Q1[C1] and Q2[C2] can still be answered more efficiently. For instance, we can compute Q1[C1] using an expression e1 = $\sigma_{rtime<t1}(\Phi_{C1}(\sigma_{rtime<t1+5}(R1)))$. Intuitively, we know from C1's condition that in order for a read r in R1 to be deleted, we need another read by readerX that trails r by 5 minutes or less. By relaxing the original condition in Q1 slightly, e1 gets enough data to remove all reads relevant to Q1. Cleansing is then applied on this slightly larger data set. Finally, e1 reapplies the original condition to remove the extra data that is needed only for cleansing.

A similar approach cannot directly be used for Q2[C2]. This is because in C2, two duplicate reads can be arbitrarily far apart in time (remember that we removed the time constraint from the original duplicate rule). Nevertheless, Q2[C2] can be alternatively answered by e2=$\sigma_{rtime>t2}(\Phi_{C2}(R2\bowtie_{epc}\Pi_{epc}(\sigma_{rtime>t2}(R2))))$, where $\bowtie_{epc}$ represents a natural join on epc between two tables, and $\Pi_{epc}$ projects the input on epc and removes duplicates. Observe that C2 only removes rows from an input sequence. Therefore, we only need to clean sequences that include at least one read satisfying the condition in Q2. The remaining sequences are not relevant because even if we clean them, no reads will be selected by Q2 anyway. That's precisely what e2 does. It first identifies sequences that have to be cleaned and then revisits R2 to extract all data on those sequences. Cleansing such a data set guarantees that all relevant anomalies are removed. Similar to e1, e2 reapplies the original condition in the end to filter out data no longer needed after cleansing. As we will see in Section 6, because the condition in typical RFID queries tends to correlate with the sequence key, the relevant sequence set can be limited effectively using this approach. It is easy to verify that e1 and e2 indeed produce the correct answer for Q1[C1] and Q2[C2], respectively.

It is hard for a conventional query optimizer to automatically generate rewrites such as e1 and e2. There are several reasons why this is challenging for a conventional optimizer. First of all, SQL/OLAP, while providing a more efficient way of bringing together different rows from the same table than self joins, hides the original row identity inside scalar aggregates and Boolean expressions (as we have seen in Section 4.3). This makes it difficult for the optimizer to do effective transitivity analysis. Second, a cleansing rule may have multiple equivalent SQL/OLAP implementations. It is difficult for an optimizer to recognize that such different query representations originate from the same rule logic and then to apply the same rewrite.

Instead of enhancing a conventional optimizer, we built a query rewrite unit outside the engine that takes a set of cleansing rules and a user query, and generates a rewritten query that gives the correct answer with respect to those rules. Because our rewrite unit is at the rule level, it can transform queries more effectively than a DBMS optimizer. Next, we will describe two styles of query rewrites, expanded (e.g., e1) and join-back (e.g., e2).

## 5.2 Expanded Rewrite

In this Section, we describe the expanded query rewrite. We assume for now that only a single cleansing rule C is defined on table R.

**Definition 1**. The pattern in a cleansing rule C specifies two types of data references: a *target* reference and a *context* reference. The former is the reference used in the action part of C (a rule has only one target reference). The rest of the references in the pattern are the context references. For example, in Figure 3(a) and Figure 3(b), references A and F are target references while references B and E are the context references (underlined).

A target reference T and a context reference X both refer to row sets in table R, but they are not independent. We refer to a condition that links T and X as a *correlation condition*. Some correlation conditions are given explicitly in the rule condition and some others are implied in the rule pattern specification (more on this later). Consider a user query Q given by $\sigma_s(R)$. Because we only need to clean data that the query cares about, Q essentially binds T to a row set $R_T=\sigma_s(R)$. Through a correlation condition cr, the context reference X is in turn bound to another row set $R_X$ referred to as the *context set* ($R_T$ and $R_X$ may overlap). Intuitively, $R_X$ is the set of rows required in order to determine whether to take any action on some rows in $R_T$. The key is to select from R, not only the query data $R_T$, but also the context set $R_X$, so that all necessary cleansing can be done. The direct pushdown approach in the previous section fails because it ignores data present only in the context sets.

If we bind query condition s to the target reference T and then run a transitivity analysis on s and the correlation condition between T and X, we may derive a new condition referencing only the context reference X. The new condition essentially defines the context set for X and we refer to it as a *context condition*. We can then use the context condition together with s to limit the amount of data extracted from R for cleansing. A correlation condition between T and X includes all conjuncts in the rule condition that refer to both T and X, as well as conjuncts that are implied in the rule pattern. There are two types of implied conjuncts, one on the cluster key *ckey* and another on the sequence key *skey*. Both *ckey* and *skey* are given in the rule definition and they are typically bound to columns epc and rtime, respectively. Since both X and T refer to rows within the same sequence (defined by *ckey*), this implies a conjunct X.*ckey*=T.*ckey*. If X is listed before (after) T in the pattern, another conjunct X.*skey*<T.*skey* (X.*skey*<T.*skey*) is implied. Adding both types of implied conjuncts for transitivity analysis allows a stronger context condition to be derived.

For context references (referred to as *position-based*) without a * in the rule pattern, there is actually a third implied correlation conjunct, on sequence position (s*pos*). For example, from the pattern in rule C2 in Figure 3, a conjunct E.s*pos*=F.s*pos*-1 is implied and it is stronger than E.*skey*<F.*skey*. Such a conjunct does not exist between context reference B (with a *) and target reference A in rule C1 in Figure 3, because the exact relative position of B to A is not important. Dealing with the implied conjunct on s*pos* is subtle. The main difficulty comes from the fact that sequence positions typically are not materialized in the input data, but are computed on the fly. Thus, when determining the context condition, we have to be careful to not change the relative position of selected rows. Before describing the solution, we first introduce the following definition and observation.

**Definition 2**. Consider a correlation condition cr between a target reference T and a context reference X. We say that cr is *position-preserving* if for any given row r referred to by T, the context set V (computed through cr) for r has the following property: for every row v in V, all rows between v and r in the original sequence also belong to V. Observe that within any data set from R containing both V and {r}, the sequence position of any row in V relative to r is the same as that in R.

**Observation 1.** (a) The following correlation conditions between a target reference T and a context reference X are position-preserving: (1) X.*ckey* = T.*ckey*; (2) X.*skey* < T.*skey* and X.*skey* > T.*skey* – t, if X is before T in C's pattern; X.*skey* > T.*skey* and X.*skey* < T.*skey* + t, if X is after T in C's pattern, where t is a positive constant. (b) Any correlation condition on columns other than *ckey* and *skey* is not position-preserving.□

*Reason:* (a) Follows from definition. (b) For any such correlation condition cr, it is always possible to construct a counter example by making the column used in cr independent of the sequence key.

Therefore, for position-based context references, we do not include all conjuncts in the correlation condition for transitivity analysis. Instead, we keep only those that are position-preserving. We are now ready to summarize the process of generating an expanded rewrite for Q[C] in Figure 4. From line 2 to line 10, we iterate through each context reference X in rule C. Depending on whether X is position-based or not, we prepare accordingly the correlation condition between X and the target reference T, as a

Inputs: s, a condition on R in a user query Q
C, a cleansing rule on R
Output: $Q_e$ , an expanded rewrite for answering Q[C]
Method: cr: correlation condition, cc: context condition
ec: expanded condition
1. cc = { }
2. for each context reference X in C {
3. cr = conjuncts in C's conditions referring to X + implied conjuncts on *ckey* and *skey*
4. if X is position-based context
5. keep only position-preserving conjuncts in cr
6. run transitivity between cr and s (bind s to target reference T)
7. d = derived conjuncts referring to X only
8. if (d not empty) cc = cc ‖ d
9. else { cc = { }; break}
10. }
11. if (cc not empty) {
12. ec = s‖cc; s' = s – cc; $Q_e = \sigma_{s'}(\Phi_c(\sigma_{ec}(R)))$ }
13. else $Q_e$ = null (no possible $Q_e$)

**Figure 4: Algorithm for Expanded**

list of conjuncts. We then apply transitivity analysis between the correlation condition rc and the query condition s (s is bound to T). If any conjunct can be derived referencing X only, it is added to the context condition cc. If there are multiple context references, context conditions are or-ed together to select the combined context sets. If any context condition cannot be derived, we set it to empty and break out of the "for" loop. From line 11 to line 13, if the context condition is not empty, we proceed to generate the expanded rewrite $Q_e$. We first compute an *expanded condition* ec as s‖cc and it becomes the predicate that can be pushed to R directly. After performing cleansing on the data set selected by ec, we have to apply s again to remove rows in the context set that we no longer need. We use an optimization here to simplify s to s', by avoiding reapplying conjuncts in s already covered in the context condition. Finally, $Q_e$ is given by an expression $\sigma_{s'}(\Phi_c(\sigma_{ec}(R)))$.

**Theorem 1.** $Q_e$ computed by the algorithm in Figure 4 gives the correct answer to Q[C].

*Proof:* (sketch) We can show that for each row r not selected by ec, r is not needed either directly by Q, or indirectly in order to clean any row of interest to Q. Therefore, not selecting r does not change the query semantics.□

We illustrate the algorithm in Figure 4 using our running examples. The first example is shown in Figure 3(c) and is based on rule C1 and query Q1. Because context reference B is not position-based, all four conjuncts (listed as cr1), including implied ones, that correlate B to target reference A can be used for deriving the context condition. The condition specified in query Q1 is given by s1, only now bound to A. By computing transitivity on s1 and cr1, the context condition cc1 includes a newly derived conjunct B.*rtime*<t1+5min and another one directly from cr1. The expanded condition is given by ec1, which can be relaxed to *rtime*<t1+5min (which is actually used in e1 in Section 5.1), if the second conjunct is not very selective.

Figure 3(d) describes the second example based on rule C2 and query Q2. Since C2 has a position-based context reference E, only

the two position-preserving conjuncts can be used as the correlation condition (listed as cr2). However, no conjuncts can be derived on E through transitivity analysis between cr2 and s2. Therefore, the expanded rewrite is not feasible for Q2. In Section 5.3, we will discuss how to handle Q2 using the join-back rewrite.

**Join Query Support**: When Q contains joins of $\sigma_s(R)$ to other tables, in general, we have to generate $Q_e$ for $\sigma_s(R)$ first and then join $Q_e$ with the rest of tables. However, for a certain class of queries, it is possible to do the joins before cleansing. Consider a query Q of the form $\sigma_s R \bowtie_{K1} \sigma_{S1} D_1 \bowtie \ldots \bowtie_{Kn} \sigma_{Sn} D_n$, where table R joins each table $D_i$ on column Ki and all joins are n to 1 (queries of this type are common since the reads table is typically only joined with other reference tables). We can convert each join condition to a conjunct of $R.K_i$ in (select $K_i$ from $D_i$ where $S_i$) such that it looks like a local condition on R. We can then apply the algorithm in Figure 4 as before. After the transitivity analysis, some of those "in" conjuncts are derived on the context reference and are added to the context condition. We refer to those conjuncts as $P_i$ on tables $D'_i$, for i from 1 to $m \le n$. Since each $P_i$ can be converted back to a join condition, this means that each table $D'_i$ can be joined to R before we apply cleansing. While pushing local predicates before the more expensive cleansing step is always a good idea, whether to apply a join before cleansing really depends on factors such as their relative cost and selectivity. There are $2^m$ possible ways of pushing the m $D'_i$ tables before cleansing and trying them all is too expensive. Instead, we employ a heuristic that favors tables with more restrictive local predicates. Specifically, we order $D'_i$ by the selectivity of $S'_i$ ascendingly (the selectivity of each $S'_i$ can be obtained from the execution plan of the original query Q after compiling it in a DBMS). We then generate m+1 SQL statements as follows: The first statement defers all joins after cleansing. Each of the next m statements pushes one more table $D'_i$ (in selectivity order) before cleansing. All m+1 statements are then compiled by the DBMS and the statement with the cheapest cost estimate is selected as the expanded rewrite.

### 5.3 The Join-back Rewrite

When the context condition is empty, the expanded rewrite is not feasible since no conditions can be pushed before cleansing. In this section, we describe a join-back rewrite that is always applicable. The idea is to remove non-relevant sequences early such that cleansing only needs to be applied on a smaller number of sequences. Again, consider a query $Q=\sigma_s(R)$. Expression $\Pi_{ckey}(\sigma_s(R))$ defines all the sequences in R that Q is interested in, because rule C only deletes or modifies (but does not insert) rows from R. If we go back to table R and fetch all rows that belong to those sequences, we will have enough data to perform the correct cleansing. The join-back rewrite for Q[C] is given by $Q_j = \sigma_s(\Phi_c(R \bowtie_{ckey} \Pi_{ckey}(\sigma_s(R))))$. This is how we derived expression e2 to answer Q2[C2] in Section 5.1.

Even when the expanded rewrite is applicable, the join-back approach could be more efficient. The tradeoff is that the former selects more rows from R at the beginning than the latter (since ec is typically less restrictive than s), but does not need to join R a second time afterwards. Furthermore, we can take advantage of the expanded condition generated in Figure 4. Given a sequence in R, the expanded condition selects all rows in it needed for the query as well as for cleansing. Thus, during join-back, we only have to bring back rows that qualify the expanded condition. The improved join-back rewrite is given by $Q_j = \sigma_{s'}(\Phi_c(\sigma_{ec}(R) \bowtie_{ckey} \Pi_{ckey}(\sigma_s(R))))$. As an example, Q1[C1] in Section 5.1 can also be answered by a join-back rewrite $\sigma_{rtime<t1}(\Phi_{C1}(\sigma_{rtime<t1+5}(R1) \bowtie_{epc} \Pi_{epc} (\sigma_{rtime<t1}(R1))))$.

We can also extend the join-back rewrite to support join queries. Consider the same join query Q $\sigma_s R \bowtie_{K1} \sigma_{S1} D_1 \bowtie \ldots \bowtie_{Kn} \sigma_{Sn} D_n$ given in Section 5.2. We can further limit the relevant sequence set by performing a semi-join between each $D_i$ and R, using $\Pi_{ckey}(\sigma_s R \bowtie_{Ki} \sigma_{Si} D_i)$. Again, there are tradeoffs on how many semi-joins to apply before cleansing. Pushing more semi-joins reduces the amount of data to be cleaned, but increases the join overhead. We follow the heuristic used in Section 5.2 by ordering $D_i$ in ascending selectivity of $S_i$. We then generate n+1 SQL queries, pushing from 0 to n semi-joins in that order before cleansing. The query with the cheapest cost estimate by the DBMS is picked as the join-back rewrite. Finally, we compare the expanded rewrite with the join-back one and pick the rewrite with a lower cost estimate for execution.

### 5.4 Supporting Multiple Rules

We now discuss how to rewrite queries with respect to a list of cleansing rules $C_1$ to $C_n$. We assume that all rules created by an application share the same *ckey* and *skey*. Because of dependencies among rules, we have to make sure that the rules are evaluated in the order of their creation time, say from $C_1$ to $C_n$.

We first illustrate how to generate an expanded rewrite for a query Q given by $\sigma_s(R)$. For each rule $C_i$, we compute a context condition $cc_i$ using lines 1 to 10 in Figure 4. If any $cc_i$ is empty, there is no feasible expanded rewrite and we have to rely on the join-back one discussed next. Otherwise, we calculate an overall context condition cc as $cc_1 \| cc_2 \ldots \| cc_n$, which selects enough context data for all the rules. By following lines 11 to 13 in Figure 4, we can determine the expanded condition ec and the condition s' accordingly. The expanded rewrite is then given by the expression $\sigma_{s'}(\Phi_{Cn} \ldots \Phi_{C1}(\sigma_{ec}(R)))$. We omit the discussion on join queries since they can be dealt with in a similar way as described in Section 5.2. Next, extending the join-back rewrite to support multiple rules is straightforward because the elimination of non-relevant sequences is independent of the cleansing rules. Therefore, after all data in those relevant sequences are extracted, we can apply the cleansing rules $C_1$ to $C_n$ in this order.

It is easy to see that both our rewrites give the correct answer to $Q[C_1 \ldots C_n]$, because all rules are applied in the right order. An interesting question is whether we can switch the evaluation order of those rules without changing the query semantics. In general, this is a hard problem and we leave it for future work. However, we observe that switching rule order may not be very crucial for achieving better performance. One can treat each rule $C_i$ as an expensive predicate on table R. Those predicates have the characteristics that their selectivities are all high (because the number of anomalies is typically small) and their costs are comparable (because the sorting cost to produce the sequence order may be dominant). Since the optimal order for evaluating a set of expensive predicates [19] mainly depends on their selectivity and relative cost, the performance difference between different orders is likely small.

## 6. EXPERIMENTS

We experimentally validate the effectiveness of the deferred cleansing approach. Our goal is to test its scalability along three

dimensions, namely, the amount of data to be queried, the number of rules to be applied, and the number of anomalies.

### 6.1 Experimental System Design

Because there is no existing benchmark for testing RFID applications, we built an RFID data generator called RFIDGen in Java. RFIDGen simulates a typical supply chain of a retailer W that keeps RFID data over the last five years together with some related reference data. All information is stored in seven relational tables and their schema and relationships are summarized in Figure 5. The primary key of each table is underlined and the arrows represent foreign key references. The number inside the parenthesis indicates the number of rows in each table (the scale factor s will be explained later).

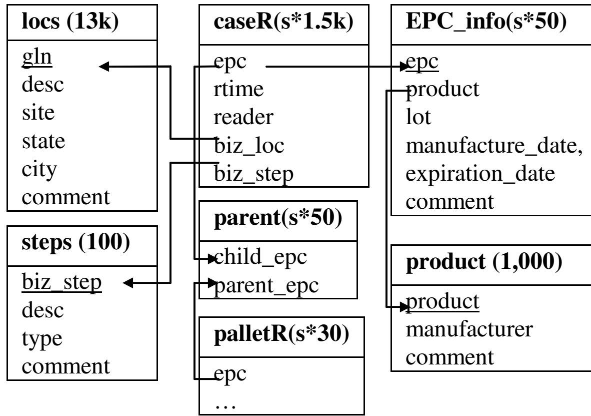


**Figure 5** RFID Data Schema

We first describe the generation of normal RFID data, i.e., without anomalies. We assume that all goods sold by W have to go through three levels of distributions: a distribution center (DC), a warehouse, and a retail store. There are 1,000 retails stores, each of which receives goods from one of the 25 warehouses, each of which in turn receives shipments from one of the 5 DCs. Every site (a DC, a warehouse, or a store) has 100 distinct locations, each equipped with an RFID reader. The "location" table stores all 13,000 distinct locations, each identified by a 13-character Global Location Number. Every shipment is read 10 times (readers randomly selected) at each of the 3 sites that it has to go through and generates a total of 30 RFID reads. The first read of an EPC is chosen randomly within a 5-year window and the latency between two consecutive reads of a shipment is randomly selected between 1 and 36 hours. Each shipment is uniquely identified by a 96-bit EPC represented by a 50-byte varchar. We assume two types of shipments, a case and a pallet. To factor in variance in goods size, we randomly choose a number between 20 and 80 as the number of cases contained in a particular pallet. For simplicity, we store case reads and pallet reads in two separate tables, "caseR" and "palletR", respectively, and store the association between a case EPC and a pallet EPC in a third "parent" table. We assume that neither the case EPC nor the pallet EPC is reused and an association entry is always valid. The schema of palletR is identical to caseR. A pallet and its associated cases always travel together and are read by the same reader within 10 minutes of each other. Each case EPC generates an entry in an "EPC_info" table, in which item specific information such as the lot number, the manufacture date and the expiration date are stored. Details about products are stored in a "product" table which is referenced by the EPC_info table. We generate a total of 1,000 different products, randomly assigned to 50 manufacturers. Each RFID read is also assigned a randomly selected business step from a total of 100 different steps stored in a "steps" table. All steps are evenly classified into 10 different types. We define the number of pallet EPCs "s" as the scale factor. For a given s, there are approximately s*50 case EPCs. Therefore, palletR, caseR, parent, and EPC_info contain s*30, s*50*30 (=s*1.5k), s*50, and s*50 rows, respectively.

We then add anomalies over the regular data. Because pallets are read more reliably than cases, we introduce anomalies to case reads only. We add five types of anomalies described in Section 4 by reversing the action of the cleansing rules. For example, if an action deletes a read from a sequence, we would add a false read that meets the rule's condition. Given an anomaly percentage D, we distributed anomalies evenly among the five different types.

**q1**. "Dwell" analysis

```
with v1 as
( select biz_loc as current_loc, rtime,
    max(rtime) over (… rows 1 preceding) as prev_time,
    max(biz_loc) over (… rows 1 preceding) as prev_loc
  from caseR where rtime <= T1 )
select l1.loc_desc, l2.loc_desc, avg(rtime-prev_time)
from v1, locs l1, locs l2
where v1.prev_loc = l1.gln and v1.current_loc = l2.gln
group by l1.loc_desc, l2.loc_desc
```

**q2**. Site analysis

```
select p.manufacturer, count(distinct s.type),
       count(distinct c.reader)
from caseR c, steps s, locs l, epc_info i, product p
where c.biz_step=s.biz_step and c.biz_loc=l.gln
   and c.epc=i.epc and i.product=p.product
   and c.rtime >= T2
   and l.site = 'distribution center 2'
group by p.manufacturer
```

**Figure 6** Benchmark Queries

| rule name | q1 | q2 |
|---|---|---|
| reader | rtime<=T1+5 min (c1) | rtime>=T2 (c4) |
| duplicate | rtime<=T1 (c2) | rtime>=T2+10min (c5) |
| replacing | rtime<=T1+20 min (c3) | rtime>=T2 |
| cycle | {} | {} |
| missing | {} | rtime>=T2 |

**Table 1** Expanded Conditions for q1 and q2

In Figure 6, we describe two representative benchmark queries and the SQL statements used in our tests. The first query q1 performs a typical "dwell" analysis that calculates the average time shipments spent between two consecutive locations. To achieve that, q1 exploits the SQL/OLAP functions to bring information of two adjacent reads of each EPC to the same row. The rest of the computation is straightforward. The second query q2 resembles a typical analytical query. Table caseR is treated as the fact table, which is joined with multiple dimensional tables to bring in the reference data. Specifically, it reports the reader utilization and business steps involved for each manufacturer at a particular distribution center. The choice of the two timestamps T1 and T2 will be explained later in this section. We use the five

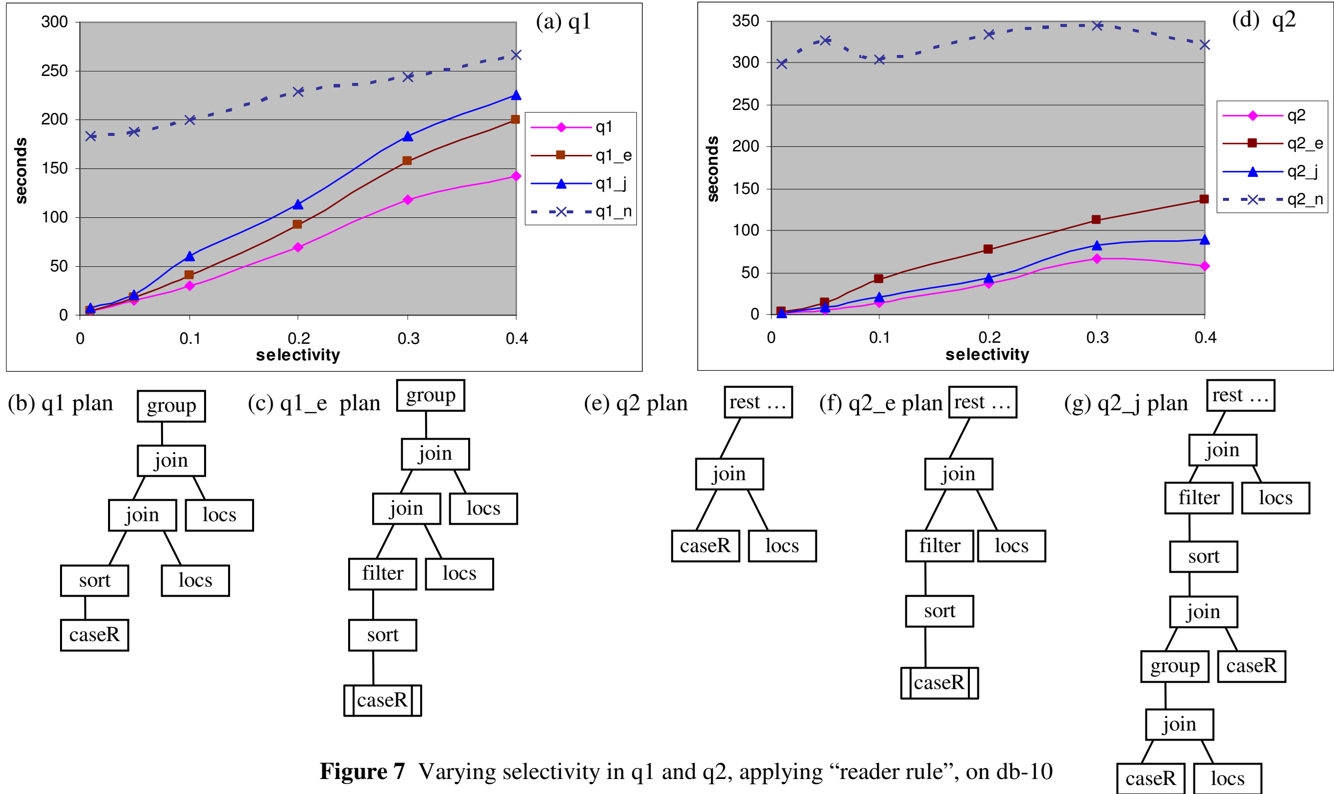


**Figure 7** Varying selectivity in q1 and q2, applying "reader rule", on db-10

cleansing rules defined in Section 4 in our experiments, with t1, t2 and t3 set to 5, 10 and 20 minutes respectively. Table 1 summarizes the expanded conditions computed by the algorithm in Figure 4 for both q1 and q2 with respect to each of the rules. Note that the cycle rule has two context references, one before and one after the target. Since neither context is bounded by time, no expanded condition exists for both queries. The missing rule has an unbounded context reference following the target and only q1 does not have an expanded condition.

We used DB2 V8.2 on a modern server-type machine running AIX for our performance evaluation. We chose a scale factor s that generates approximately 10 million normal case reads (about 1GB). We then loaded four different databases into DB2, corresponding to anomaly percentage 10, 20, 30 and 40, respectively (we refer to them as db-10 to db-40). Data is loaded in an order partially correlated with time. For each database, we indexed all columns in both caseR and palletR except for the reader column. The parent table was indexed on child_epc to provide fast case-to-pallet lookups. The rest of the tables have indexes only on their primary keys except for the locs table which is further indexed on site and the steps table is further indexed on type. Each database uses a bufferpool of 160MB. For each test, we compare the elapsed time of running a benchmark query q on the dirty data directly (referred to as q), the expanded rewrite (q_e), the join-back rewrite (q_j), and the naïve rewrite (q_n) that first cleans all data and then evaluates q. Note that unlike the other three, q does not always give the correct result, and is only used for baseline comparison. We do not explicitly compare eager cleansing with deferred cleansing in the experiments. However, the cost of eager cleansing should be comparable to that of q, since the anomaly percentage is typically small.

## 6.2 Varying Selectivity

In this section, we test deferred cleansing by scaling the data size requested by queries. We assume that only the reader rule is enabled, and use the database with 10% anomalies (db-10). We then vary the selectivity of the predicate on rtime in both q1 and q2 from 1% to 40%, by adjusting T1 and T2 accordingly.

The results for q1 are shown in Figure 7(a). Both q1_e and q1_j perform much better than the naïve version across all selectivities and q1_e is more effective than q1_j. We now take a closer look at the execution plans. The plan for q1 is shown in Figure 7(b). It first scans table caseR using the index on rtime. It then evaluates the two scalar aggregates specified by SQL/OLAP by sorting input data in (epc,rtime) order. After that, q1 joins table locs twice and does the final aggregation. In comparison, the plan for q1_e is shown in Figure 7(c). Since the predicate on rtime is expanded by five minutes, q1_e needs to bring in a little bit more data from caseR initially (shown as a box with double side edges). Next, it evaluates the reader rule, by first sorting the input on (epc,rtime) and then removes anomalies through a filter. Once cleansing is completed, it continues with the rest of q1. It is important to see that because the ordering requirement for the SQL/OLAP evaluation in q1 is the same as that in the cleansing rule, data only needs to be sorted once. Although this seems incidental, we expect such order sharing to be common in RFID applications because it is often useful to process RFID data in sequence order. Therefore, compared with q1, q1_e only incurs the extra overhead of computing one more scalar aggregate, but does not have an extra sort. This explains why q1_e adds only a small overhead on top of q1. In this example, since the only predicate restricting caseR is expandable, the expanded approach is always better than the join-back one which has to access table caseR twice. Finally,

the naïve approach has to sort all data since it does not push down any predicate, and thus is much worse.

We now move to q2 and show the performance results in Figure 7(d). Similar to q1, both q2_e and q2_j perform much better than the naïve approach. In contrast to q1, this time q2_j becomes more effective than q2_e. We explain the difference by analyzing the plans. A partial plan for the original query q2 is shown in Figure 7(e). q2 first joins table caseR and locs since they are the only ones with local predicates. The box denoted by "rest …" includes the rest of the joins (which don't reduce the cardinality further) and the final aggregation. The plan for q2_e is shown in Figure 7(f). Note that q2_e does one more sort than q2 since the ordering requirement from cleansing in SQL/OLAP is different from that for grouping. Because only the rtime predicate is expandable, the cleansing process has to be done immediately after accessing table caseR before joining table locs. Thus, q2_e has to sort data including those to be rejected later by the join. The plan for q2_j in Figure 7(g) exploits the constraint on both table caseR and locs by joining them first. It then computes a unique list of EPCs e through grouping, which essentially contains the set of shipments that q2 actually cares about. After that, q2_j visits table caseR a second time to extract the full history of EPCs in e for cleansing. Note that the site column is partially correlated with EPC since at a particular site, a given EPC is read either multiple times (since it goes through that site) or none at all (since it does not go through the site). As a result, the predicate on site, in addition to the one on rtime, helps reduce the size of E significantly. This benefits q2_j a lot since it has to sort fewer data and computes fewer scalar aggregates. To complete this discussion, q2_j still needs to apply the predicate on rtime and site a second time to remove data no longer needed after cleansing. Such an overhead is small since it is on a small data set. The rest of q2_j is the same as q2. In summary, when the selectivity of the expandable predicate is small, q2_e is comparable to q2_j since the data reduction in the latter is offset by the join-back overhead. However, as the predicate on rtime becomes less restrictive, q2_j performs much better because it can effectively exploit the filtering power from the predicate on site.

As an extreme test, we design another query q2' by swapping l..site and s.type in q2 and changing the constant from a DC to a specific business type. We deliberately populated data such that s.type is completely uncorrelated with EPCs. The results for q2' are shown in Figure 8. q2'_j is no longer much better than q2'_e. Although the predicate on s.type reduces the number of reads, it does not significantly reduce the set of EPCs (many EPCs having a single read) required for cleansing. Thus, the overhead of sorting and computing scalar aggregates in q2'_j is comparable with that in q2'_e. In reality, predicates on columns that (partially) correlate to EPCs are used more often in application queries because multiple reads for a shipment can be analyzed together. For example, it is more reasonable for q2' to select a set of business steps all common to some shipments.

### 6.3 Varying Rules and Dirty Percentage

In this section, we first study the scalability of deferred cleansing in terms of the number of rules. In both q1 and q2, we fixed the selectivity of the rtime predicate at 10% and chose the database with 10% anomalies. We then scale the number of rules from 1 to 5. The rules are added in the order as listed in Table 1. For both q1 and q2, the expanded approach is only feasible up to the first three rules. The join-back approach is valid for all rules. The results are presented in Figure 9(a) and Figure 9(b). From 1 rule to 3 rules, the increase in the best rewrite strategy q1_e and q2_j is fairly small. Observe that the ordering requirements for all rules are the same. Thus, only the first rule incurs the sorting overhead. Subsequent rules share the same sort and only pay the overhead of computing their own scalar aggregates. Starting from 4 rules, only the join-back approach becomes applicable. For both q1_j and q2_j, the cycle rule adds somewhat more overhead compared to previous ones. This is mainly because no expanded conditions can be applied on table caseR during join-back. The missing rule adds the most overhead among all rules. Unlike other ones, this rule takes input from the following derived table.

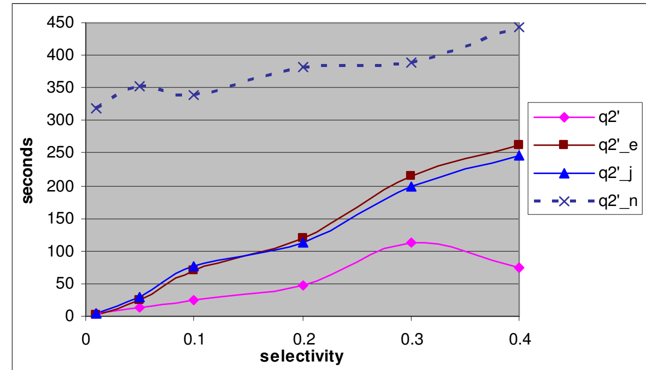


**Figure 8** Varying selectivity in q2', 1 rule, on db-10

```
select epc, rtime, biz_loc, biz_step, reader, 0 as is_pallet
from caseR
union
select child_epc, rtime, biz_loc, biz_step, reader, 1 as is_pallet
from palletR, parent
where palletR.epc = parent.parent_epc
```

Conditions in q1 and q2 are applied on both caseR and palletR to obtain the set of EPCs to be cleaned and join-back is also performed on both tables. Since approximately every case read is now paired with a pallet read, the amount of data to be sorted is now doubled. The extra joins needed to retrieve pallet reads also add some overhead, but this is secondary because the reads are restricted to a smaller set of EPCs. Nevertheless, both q1_j and q2_j perform significantly better than the naïve approach, which takes about 1,000 seconds (out of the pictures) with 5 rules.

Our last experiment tests deferred cleansing with respect to the number of anomalies. Again, we fix the selectivity of the predicate on rtime at 10% in both q1 and q2. We applied the first three rules as listed in Table 1. We then test the queries on four different databases, with anomalies from 10% to 40%. The results are shown in Figure 9(c) and Figure 9(d). For both q1 and q2, the expanded query and the join-back query increase only slightly with more and more anomalies, and match the trend of the original query. Note that X% anomalies do not translate to an X% larger database because anomalies such as missing reads actually reduce the amount of raw data.

**Summary**: Our experimental results show that deferred cleansing is affordable for typical analytic queries on RFID data. Both the expanded and the join-back approach perform much better than the naïve approach. When both are applicable, there are tradeoffs between the two. It is also important to note that the overhead of