An Approach to Detecting Relevant Updates to Cached Data Using XML and Active Databases

Essam Mansour
International University in Germany
School of Information Technology
Campus 3, D-76646 Bruchsal, Germany
essam.mansour@ieee.org

Hagen Höpfner
International University in Germany
School of Information Technology
Campus 3, D-76646 Bruchsal, Germany
hoepfner@acm.org

ABSTRACT

Client/server information systems use caching techniques to reduce the volume of transmitted data as well as response time and, especially in the case of systems with mobile clients, to reduce energy consumptions. Updating the server database might cause inconsistencies between server data and cached data. Guaranteeing consistency at least demands to invalidate outdated caches. To avoid invalidation of caches that are not affected by a particular update one must check the relevancy of each update for each cache. It has been proven, that this can only be done on a stateful server.

This paper presents the purely database system (DBS) based DRUPE method for checking the relevance of server side updates to cached data by analyzing the intersection between modified data and cached data. A non-empty intersection means that the update operations are relevant to the cached data. The necessary cache descriptions are stored in form of XML-documents inside the DBS. The paper introduces the used XML-model XRDL as well as the relevancy proof-of-concept system Uptime. The main contribution of our work is the system utilizes the DBS utilities to detect update relevancy, notify clients and manage the required repository of the queries issued by the clients. Hence, no additional middleware is required in order to realize consistency aware client/server information systems, even if clients are small footprinted mobile devices.

1. INTRODUCTION

Data caching is an appropriate technique for reducing the volume of transmitted data and response times in distributed systems in general and in client/server information systems in particular. If clients are mobile devices such as mobile phones or embedded devices that use wireless communications, optimizing data transmissions also increases the uptime of the clients by decreasing their energy consumptions [2, 10]. However, the major drawback of caching techniques, which per se create redundant data on (mobile) clients, is the potentiality of inconsistencies. Server side updates must also update the cached copies or at least invalidate them. Especially in information systems with many clients, such as mobile information systems, it is not useful or even impossible to invalidate all caches for each server side update. Hence, it is necessary to identify only those client caches that are affected by the update.

As proven in [9] checking update relevance in general requires the usage of a stateful information systems server that stores the relationships between (mobile) clients and data cached by them. Clients retrieve the data by issuing database queries. So, the server can keep this semantic cache information and maintain an index [8] that represents information about which client caches which part of the database. As shown in [7, 11] it is then possible to check the relevance of server side updates using the index and to notify only the affected clients. So far, relevance checks are done within a middleware component on top of the database management systems. This approach causes unnecessarily complex information systems.

Utilizing DBSs to detect relevant updates to cached data and notify clients by such updates leads to avoiding several applications layers and reducing the code complexity. The development of an approach to detecting update relevancy as a DBS built-in function is the main topic to be investigated in this paper.

This paper presents the DRUPE (Detecting Relevant Update Easily) method for checking the relevance of the manipulation operations insert, delete and update over multi-set semantics of the relational data model. The main objective of this method is to test the update relevancy using queries of relational algebra that check the intersection between the cached data and the modified data. A manipulation operation is to be irrelevant to the cached data, if the intersection is empty. Otherwise, this manipulation operation is relevant. We introduce an XML-based model XRDL (XML-Based Relational Algebra) for storing queries issued by clients and the manipulation operations executed by the server. It provides XML representation for the queries and manipulation operations. This XML representation is to be stored as XML documents in modern DBSs that must provide XML management support, such as DB2 [12] and Oracle [15]. The paper highlights a proof-of-concept system, called Uptime (Update Notification Made Easy) that utilizes the DRUPE method and the XRDL model to develop an update notification mechanism as built-in function inside DBSs that provides XML management support and triggering mechanism.

The remainder of this paper is organized as follows: Section 2 discusses the related work. Section 3 presents an application example used through the paper to illustrate our ideas. Section 4 gives an overview of the used query notation. Section 5 describes the DRUPE method. Section 6 outlines the XRDL model. Section 7 introduces the Uptime system. Section 8 presents first experimental results and discusses the scalability and performance of Uptime. Section 9 concludes the paper and gives an outlook on future research.
2. RELATED WORK
Finding irrelevant updates strongly overlaps with the theory of incremental view updates [1]. In fact, cached data can be considered to be a (materialized) view over a global database. Many algorithms have been developed in order to check the relevance or irrelevance of modifications to the global DB by comparing the queries (views) to a query that would result in the updated tuples on a semantic level. There are two major drawbacks with these approaches: language limitations and the empty set problem.

The algorithms utilize the query containment problem (QCP) [3]. Therefore, they have similar limitations. In [17] it was shown that the QCP is undecidable for arbitrary calculus queries, for arbitrary queries in the relational algebra, and for logical query languages [16]. However, [3] gives the proof that QPC is decidable but NP-complete for conjunctive queries. Also, other subsets of these conjunctive queries were researched, and there are subsets with better complete for conjunctive queries. Therefore, they have similar limitations. In [17] it was shown that the QCP is undecidable for arbitrary calculus queries, for arbitrary queries in the relational algebra, and for logical query languages [16].

The empty set problem [9] results from the fact that QPC is defined on the result sets: a query \( Q_2 \) contains a query \( Q_1 \), if, for each database state, the result of \( Q_1 \) is a subset of the result of \( Q_2 \). From the set theory we know, that the empty set is a subset of every set. Therefore, if for example a delete would not delete anything (e.g. the tuples that should be deleted are not in the relation), then the result would be an empty set (that is contained anyway). The system would notify the client about an update that did not change anything.

Besides these more general researches there exist papers dedicated to the incremental view update problem. [13] consider inserts and deletes in combination with horizontal database fragments. Therefore, they do not allow projections. Inserts, deletes and modifications are considered in [1]. However, the approach is limited to equal-joins and do not support self-joins. Algorithms that use logical query languages typically forbid negations [4].

The solution for overcoming the limitation of a purely semantic relevance check is to analyze the database extension and not only the database intension. In [9] we introduced an approach that calculates test queries based on the query predicates and the modification operations. The test queries are executed on the database. Their result sets show the relevance or irrelevance of the update for the whole query. By splitting the queries into predicates it is possible to optimize the relevance check for systems with many partially overlapping queries by testing them in parallel. However, the tests in [9] are based on the set semantics (SELECT DISTINCT) of the relation algebra that is “uncommon” in most application scenarios. Some ideas on testing update relevance under multiset-semantics have been published in [7]. Furthermore, [6] show, that the test query idea can also be used for handling update relevance checks of context aware queries. However, all these approaches implement the update relevance test as middle-ware component but do not utilize the services, functions and build-in features of modern database management systems.

3. AN APPLICATION EXAMPLE

The cinema database introduced in this section is used as an application example to demonstrate the ideas of our work and evaluate the UptiME system. The cinema database as shown in Figure 1 conceptually consists of four entities, cinema, auditorium, location and movie. In this paper, the used queries and manipulation operations are applied to the tables representing the entities cinema and location. These tables are cinema_tab and location_tab. The tables in Figure 2 show sample data of cinema_tab and location_tab respectively. As they show, there are two cinemas belong to Karlsruhe and other two belong to Bruchsal.

4. QUERY REPRESENTATION

In mobile information systems, applications generate queries and send them to the server. Therefore, there is no need to support descriptive query languages, such as SQL. Queries are to be represented in a useful way for storage and retrieval. The relational algebra representation [5] is an efficient way to represent queries over data stored in relational database.

The query noted in this paper is the notation of the relational algebra operators, such as selection (\( \sigma \)), join (\( \bowtie \)), projection (\( \pi \)) and rename (\( \rho \)). Assume a mobile client issued the query QCL to know the name, street, and hotline of cinemas in Karlsruhe, whose postal code is 76131, where the rate of the cinema is greater than four. Figure 3 shows a relational algebra of QCL.

A relational algebra query tree might have several equivalent relational algebra trees due to the use of algebraic properties for query optimization [5]. For example, the relational algebra expression of the query QCL, shown in Figure 3, has the following predicates:

- a selection predicate (SP), such that \( ctab.RATE \) is one of the attributes of the previously renamed relation cinema_tab, and \( ltab.Postal_Code \) is an attributes of the also previously renamed relation location_tab.

- a join predicate (JP), such that \( ctab.LID \) is an attribute in cinema_tab and \( ltab.LID \) comes from location_tab.

\[ SP \left( \sigma_{ctab.cname = \text{"Cineplex"}} \left( \sigma_{ctab.RATE > 4} \left( \ast_{ctab.LID = \text{"101"}, ltab.Postal_Code = \"76131\")} \left( ctab.cname, ctab.LID, ctab.HOTLINE, ctab.RATE, ctab.RENEWED_ON \right) \bowtie_{ctab.cname = \text{"Cineplex"}} \left( \ast_{ltab.LID = \text{"101"}, ltab.Postal_Code = \"76131\")} \left( ltab.LID, ltab.PLACE, ltab.STREET, ltab.POSTAL_CODE \right) \right) \right) \]

Figure 3: The relational algebra of QCL.
• a projection predicate (PP), such that \( ctab.cname \) as well as \( ctab.hotline \) are attributes of the relation \( cinema_tab \), and \( ltab.street \) is an attribute of \( location_tab \).

In relational algebra, selection and projection operators could be pushed inside a join operation under certain condition [5]. We can push a selection inside a join, since it involves only attributes of one relation. Moreover, pushing a projection operation inside a join requires that the result of the projection contain the attributes used in the join. Figure 4 shows an equivalent relational algebra expression for the one shown in Figure 3 according to the algebraic properties for query optimization.

\[
\begin{align*}
\forall & \text{ctab.cname, ctab.hotline, ctab.LID} : (\text{ctab.RATE} > 4) & \rightarrow & \{\text{ctab}('cinema_tab')\} \\
\forall & \text{ltab.LID} : \text{ltab.LID} \\
\forall & \text{ltab.street, ltab.LID} : (\text{ltab.Postal_Code} = '76131') & \rightarrow & \{\text{ltab}('location_tab')\}
\end{align*}
\]

Figure 4: An equivalent relational algebra for the QCL query.

In order to support our method for detecting relevance update, it is more efficient to store queries in the form of a set of relations restricted to specific rows and attributes and a set of join predicates. Therefore, it is assumed that queries are to be generated by an application in such form. Generally, any SQL query could be mapped into relational algebra expression, such that the selection and projection operation are pushed inside the join operation. Consequently, we can easily utilize our method with several mobile information systems, in which relational data is queried by mobile clients.

5. DRUpE: A METHOD FOR CHECKING UPDATE RELEVANCE

The DRUpE \(^7\) detects the relevance of insert, delete and update operations over multiset semantics of the relational data model. The main idea is to check the intersection between modified data and cached data, which is a result of specific queries. A non-empty intersection means that the update operations are relevant to cached data. The method retrieves the data of the intersection using a query/queries constructed from the manipulation operations and the queries, whose result is cached on at least one client.

5.1 Test the relevance of inserts

We use the SQL insert construct \( \text{INSERT INTO relation} \) (column1, [column2, ...]) \VALUES (value1, [value2, ...]) for adding a new tuple into a relation. The number of columns and values must be the same. If a column is not specified, the default value for the column is used.

![Data to be cached after insertion (D1)](image)

Figure 6: The effect of an insert operation on cached data.

Figure 6 illustrates the effect of an insert statement on the cached data. That effect is represented by the intersection \( D1 \) between the inserted data and the cached data. The data that belongs to the intersection \( D1 \) is data to be considered in the result of queries issued previously by mobile clients. Therefore, the insertion is a relevant manipulation operation, if the intersection \( D1 \) is not empty. Consequently, the mobile clients should be notified to update their cached data. The intersection is to be empty if and only if the new inserted tuple is matching the selection and join predicates of a query regardless\(^3\) the content of the projection predicate.

Assume the insert statement \( \text{MO1} \), shown in Figure 5, is to be executed on the server. The intersection \( D1 \) could be retrieved using a relational algebra query constructed from the insert statement, \( \text{MO1} \), and a query whose result is cached on at least one mobile client. For example, the intersection \( D1 \) of the data inserted by \( \text{MO1} \) and the cached data produced by the query \( \text{QCL} \) could be retrieved by the query \( \text{QUs} \):

\[
\sigma_{\text{ltab.Postal_Code} = '76131'}(\text{ltab}(\text{location_tab}))
\]

The selection predicate in \( \text{QUs} \) is constructed as follows:

- 7 > 4 results from the selection predicate of the query \( \text{QCL} \) by replacing the attribute \( \text{RATE} \) with its corresponding value in the inserted tuple of the operation number \( \text{MO1} \) shown in Figure 5.
- \( 102 = \text{ltab.LID} \) is constructed from the join predicate of \( \text{QCL} \) by replacing the attribute \( \text{ctab.LID} \) with its corresponding value in the inserted tuple of the operation number \( \text{MO1} \) shown in Figure 5.
- \( \text{ltab.Postal_Code} = '76131' \) is the rest of the selection predicate of \( \text{QCL} \) and associated with the un-manipulated relation(s), in this case \( \text{location_tab} \).

The general algorithm to check whether the relevance of an insertion is as follows: If the modified relation \( MR \) was not queried by at least one client, then this insertion is not relevant to cached data. Otherwise for each query \( CQ \) retrieving data from \( MR \) do:

1. if the attributes of the selection predicate or the join predicate do not have a value in the insert statement, then this insertion is not relevant to cached data.
2. else construct from the selection predicates of \( CQ \) new selection predicates \( \text{NSP1} \) by replacing the attributes with their corresponding value in the insert statement, then map the join predicates of \( CQ \) into selection predicates \( \text{NSP2} \) by replacing the attributes with their values in the insert statement.

\(^3\)This only holds for the multiset semantics of the relational algebra.

\(^7\)the acronym stands for Detecting Relevant Update Easily
3. construct a query by removing \( MR \) from the original query issued by a mobile client and replacing the selection and join predicates related to \( MR \) with the new selection predicates \( NSP1 \) and \( NSP2 \) to check the intersection \( D1 \).

4. if the result of the constructed query is non-empty result, return the ID of the client who issues the query \( CQ \).

5.2 Test the relevance of deletes

We use the SQL delete construct `DELETE FROM relation [WHERE condition]` for removing rows from a relation. Any row that matches the WHERE condition will be removed from the relation.

Figure 7 illustrates the effect of a delete statement on the cached data. That effect is represented by the intersection \( (D1) \) between the deleted data, which matches the WHERE clause \( WClause \) and the cached data, which represents queries results. The data that belongs to the intersection \( D1 \) is data to be removed from the result of queries issued previously by mobile clients. Hence, the deletion is a relevance update operation, if the intersection \( D1 \) is not empty. Consequently, the mobile clients should be notified to update their cached data. The intersection is to be not empty if and only if the deleted rows match the selection \( (SP) \) and join \( (JP) \) predicates of a query regardless the content of the projection predicate. The data of this intersection is to be retrieved by a query that picks rows matching the \( WClause \) of the deletion and the predicate \( SP \) and \( JP \) of the query.

Assume the delete statement \( MO2 \), shown in Figure 5, is to be executed on the server. The intersection \( D1 \) could be retrieved using a relational algebra query constructed from the delete statement and a query whose result is cached on at least one mobile client. For example, the intersection \( D1 \) of the data deleted by \( MO2 \) and the cached data produced by \( QCL \) could be retrieved by the query \( QD \):

\[
\sigma_{ctab.CID = 9903 \land ctab.RATE > 4 \land ctab.Postal_Code = \text{'76131'}}
\]

\[
\rho_{\text{cinema\_tab}}(\text{P1}(\text{ctab.LID} = \text{ltab.LID} \rho_{\text{location\_tab}}(\text{P2})))
\]

The query \( QD \) consists of the selection predicates \( P1 \) and \( P2 \), and a join predicate \( \text{P1}(\text{ctab.LID} = \text{ltab.LID}) \) such that: \( P1 \) is the WHERE clause of the delete statement, \( P2 \) and the join predicate are the selection predicate and join predicate of the query \( QCL \).

Before deleting the rows matching the WHERE clause, we check whether the result of the query \( QD \) is empty or not. If the result is not empty, then the deletion is a relevant manipulation operation, and the clients issuing the queries, which retrieve data from the modified relation, should be notified.

The general algorithm to check whether a delete operation is relevant to cached data or not is as follows:

- if the modified relation \( MR \) was not queried by at least one client, then this deletion is not relevant to cached data.
- else if the delete statement does not have a WHERE clause, then this deletion is a relevant manipulation operation,

- else for each query \( CQ \) retrieving data from the relation \( MR \)

1. construct a query by adding the WHERE clause of the delete operation to the selection predicate of the query \( CQ \) to check the intersection.

2. if the result of the constructed query is non-empty result, return the ID of the client who issues the query \( CQ \) to be notified.

5.3 Test the relevance of updates

A SQL update statement changes data of one or more rows in a relation. Either all the rows can be updated, or a subset may be chosen using a condition. The update statement has the following form:

\[
\text{UPDATE relation SET columnName = value [, columnName = value ...] [WHERE condition]}
\]

Figure 8 illustrates the effect of the previous case. That effect is represented by the intersections \( (D1 \) and \( D2 \)) between the updated data, which matches the WHERE clause \( WClause \), and the cached data, which represents queries results. The data belongs to the intersection \( D1 \) is data to be updated by new values, and this data is part of a result of queries issued previously by mobile clients. The data belongs to the intersection \( D2 \) is data to be updated by the same values, and this data is part of a result of queries issued previously by mobile clients. Therefore, the update operation in this case is a relevance update operation, if the intersection \( D1 \) is not empty. Consequently, the mobile clients should be notified to update their cached data. The intersection \( D1 \) is to be not empty if and only if the updated rows:

- match the selection \( (SP) \) and join \( (JP) \) predicates of a query,
- and
- are modified with new values.

The data of the intersection \( D1 \) is to be retrieved by a query that picks rows matching the \( WClause \) of the update operation and the selection predicate \( SP \) and \( JP \) of the query, and the updated attributes are to be modified by new values.

Assume the update statement \( MO3 \), shown in Figure 5, is to be executed on the server. The \( MO3 \) statement is an update statement that modifies the attribute \( HOTLINE \), which is projected in the query \( QCL \) shown in Figure 3. The intersection \( D1 \) could be retrieved using the relational algebra query \( QU \) constructed from the update statement and a query whose result is cached on at least one mobile client. The result of the query \( QU \) is to be checked before executing the update operation. For example, the intersection \( D1 \) of the data updated by \( MO3 \) and the cached data produced by the query \( QCL \) could be retrieved by the query \( QU \):
In category D2 there is no need to modify it.

\[ \sigma_{\{\text{ctab.cid}=90902 \land \neg (\text{hotline}='0721-2059-3333')\}}(\pi_{\text{ctab.RATE}}(\text{cinema_tab}) \bowtie_{\text{ctab.LID}=\text{tab.LID}} \text{π}_{\text{tab}(\text{location_tab})}) \]

QU is produced by adding the selection predicates (ctab.cid = 90902 ∧ \neg (\text{hotline}='0721-2059-3333')) to OCL, such that: ctab.cid = 90902 is the WHERE clause of the update statement, and \neg (\text{hotline}='0721-2059-3333') means that the value of the attribute HOTLINE is to be changed to a new value, as the query QU is to be executed before the update operation.

The update operation should be considered as a relevant manipulation operation if and only if categories D1 or D3 have occurred. In category D2, the update should not be considered as a relevant update, because the data is already cached at the client side, and there is no need to modify it.

Assume the update statement MO4, shown in Figure 5, is to be executed on the server. It modifies the attribute RATE, which is used in a selection predicate of the query OCL shown in Figure 3.

The intersection D1 could be retrieved using a relational algebra query constructed from the update statement and a query whose result is cached on at least one mobile client. For example, the intersection D1 of the data updated by MO4 and the cached data produced by the query OCL could be retrieved by the query QU2:

\[
\sigma_{\{\text{ctab.renewed_on}=1999\}}(\pi_{\text{ctab R}}(\text{cinema_tab}) \bowtie_{\text{ctab.LID}=\text{tab.LID}} \text{π}_{\text{tab}(\text{location_tab})})
\]

The query QU2 is constructed by adding the following selection predicates to OCL:

- (ctab.renewed_on = 1999) is the WHERE clause of the update statement.
- ((ctab.RATE > 4) ∧ \neg (7 > 4)) means this row will not be a part of the query result after the update, but it was a part of the result before the update, category D1. The value 7 is the new value assigned to the attribute RATE by MO4 and
- (\neg (ctab.RATE > 4) ∧ (7 > 4)) means this row was not a part of the query result before the update and will be a part of the query result after the update, category D3,
- the previous two predicates for D1 and D3 are added as a disjunction composite predicate, (D1 or D3).

Regardless of the existence of case D2 the relevancy will not be affected, but it is important to check that data is only in categories D1 or D3 as it has been detected in the previous selection predicates.

Assume the update statement MO5, shown in Figure 5, is to be executed on the server. The statement is an update statement that modifies the attribute LID, which is used in a join predicate. The intersection D1 between the data updated by MO5 and the cached data produced by the query QU could be retrieved by the query QU3:

\[
\begin{align*}
\text{OQ} & \leftarrow \sigma_{\text{ctab.RATE} > 4, \text{ctab.Postal_Code} = '76131'}(\pi_{\text{ctab}(\text{cinema_tab})} \bowtie_{\text{ctab.LID}=\text{tab.LID}} \text{π}_{\text{tab}(\text{location_tab})}) \\
\text{IDV} & \leftarrow \pi_{\text{CID}, \text{cinema_tab.LID}}(\text{OQ}) \\
\text{IDN} & \leftarrow (\sigma_{\text{tab.R} < 2000}(\text{π}_{\text{tab}(\text{cinema_tab})}) \bowtie_{\text{tab.CID}=\text{tab.ID}}(\text{IDV})) \\
\text{VN} & \leftarrow \sigma_{\text{tab.LID} < 101}(\text{IDV}) \\
\text{D1} & \leftarrow \pi_{\text{cinema_tab.LID}}(\text{IDN}) \cup \pi_{\text{VN}, \text{cinema_tab.LID}}(\text{VN})
\end{align*}
\]

OQ is the selection and join predicates of the original query, IDV is the ids of the cinema picked in the query result and the LID values participated in the join. IDN contains the rows satisfy the update where clause and exist in the result of the query. If VN is nonempty that means the new value, 101, assigned to the attribute used in the join predicate is not in the values succeed to join. D1 will contain any updated row, which was considered in the query result, and because the new value is not one of the values joining with the rows from other table(s) the row is to be removed from the query result.

Moreover, the intersection D3 of the data updated by MO5 and the cached data produced by the query QCL could be retrieved by the query QU4:

\[
\begin{align*}
\text{NIN} & \leftarrow \pi_{\text{CID}, \text{cinema_tab.LID}}(\text{cinema_tab}) - \text{IDV} \\
\text{CRW} & \leftarrow \sigma_{\text{tab.R} < 2000}(\text{ctab.RATE} > 4)(\text{cinema_tab}) \\
\text{VA} & \leftarrow \sigma_{\text{tab.LID} < 101}(\text{IDV}) \\
\text{WN} & \leftarrow \text{CRW} \bowtie_{\text{CRW.CID}=\text{WN.CID}}(\text{NIN}) \\
\text{D3} & \leftarrow \pi_{\text{cinema_tab.LID}}(\text{WN}) \cup \pi_{\text{VA}, \text{cinema_tab.LID}}(\text{VA})
\end{align*}
\]

NIN contains the rows that were not picked in the query result. CRW contains the updated rows that at the same time satisfy the selection predicates of the query associated to the updated relational. In this example there is only one selction predicate related to cinema_tab, which is \text{RAT} > 4. If VA is nonempty that means the new value, 101, assigned to the attribute used in the join predicate is already in the values succeed to join. WN contains the updated rows that were not in the query result and at the same time satisfy the selection predicates of the query over the updated relation. D3 contains the updated rows that are to be considered in the query result after the update because the new updated value will join these rows with the rows picked by the rest of the query.

The general algorithm to check whether an update operation is relevant to cached data or not is as follows:

- if the modified relation MR was not queried by at least one client, then this update operation is not relevant to cached data.
- else if the update statement does not contain any attribute used in a selection, join, or projection predicate of a query, whose result is cached on at least one mobile client, then this update is irrelevant update operation.
- else if the update statement contains only attribute(s) used in a projection predicate, then for each query KQ:

![Figure 9: The effect of updating a selection or join attribute.](image-url)
1. construct a query by adding the WHERE clause of the update operation and the negation of the update SET clause to the selection predicate of $KQ$ to check the intersection $D1$.

2. if the result of the constructed query is not empty, return the ID of the client who issued $KQ$ to be notified.

   • if the update statement contains attribute(s) used in a selection predicates, then for each query $KQ$:
     1. construct a query, whose selection predicates are the selection predicates matching category $D1$ or $D3$.
     2. if the result of the constructed query is not empty, return the ID of the client who issued $KQ$ to be notified.

   • if the update statement contains attribute(s) used in a join predicates, then for each query $KQ$:
     1. construct two queries, whose selection and join predicates are matching category $D1$ or $D3$, respectively.
     2. if the result of at least one of the constructed queries is not empty, return the ID of the client who issued the query $KQ$ to be notified.

6. XReAl: AN XML-BASED MODEL FOR QUERIES AND MANIPULATIONS

XReAl provides an XML-based model for queries issued by mobile clients and manipulation operations, which are to be executed by the server. The XReAl model consists of three main components, mobile client, query and manipulation. The mobile client component represents a particular mobile client and its contextual information. The details of the mobile client component is outside the scope of the paper. The query component represents a specific query issued by a mobile client in the form of relational algebra, as discussed in Section 4.

The XReAl specifications are to be stored and retrieved using modern DBSs, which is utilized to manage the data at the server. That means on the one hand the management of mobile queries and relevant functions, such as detecting update relevancy, is to be integrated into and supported by the relational DBS. On the other hand, the mobile query management is moved from the application layer to the database layer. The XReAl specifications for queries and manipulation operations assist in developing a mechanism for detecting the update relevancy and notifying mobile clients as a DBS built-in function. This mechanism generates the test queries formalized by the DRUP$E$ method by querying the XReAl specifications.

6.1 The XReAl Model for Queries

The XReAl model formalizes a relational algebra query as a query element that consists of two attributes, QID and MCID, and a sequence of elements, relations and joins. Figure 10 shows the XML schema of the query component. The QID attribute represents a query identification. The MCID attribute represents the identification number of a mobile client that issued the query. A query might access only one relation. Therefore, a query element contains at least one relations element and might has a joins element.

The relations element is composed of a sequence of at least one relation element. The relation element consists of an identification attribute, called RID, and a sequence of elements, name, rename, selections and projection. The name element represents the name of the relation. The rename element denotes the temporally name used to refer to the relation in the query. The selection element is composed of a sequence of a spredicate element of type predicateUDT. The projection element consists of a sequence of at least one attribute element of type attributeUDT.

The predicateUDT type is a complex type composed of one of the elements simplePredicate or compositePredicate, as depicted in Figure 11. The simplePredicate element consists of a sequence of elements, attribute, operator and operand. The attribute element is of type attributeUDT.

The operator element is of type logicalOperatorUDT, which is a simple type that restricts the token datatype to the values (eq, neq, leq, lt, geq, gt, and gteq). Respectively, they refer to equal, not equal, less than, less than or equal, greater than, and greater than or equal. The operand element is composed of one of the elements value or attribute. The value element is to be used with selection predicates. The attribute element is to be used with join predicates.

The compositePredicate element consists of a sequence of elements, rpredicate, junction and lPredicate. The rpredicate and lPredicate elements are of type predicateUDT. Consequentially, the rpredicate and lPredicate elements might consists of simple or composite predicate. The junction element is of type junctionUDT, which is a simple type that restricts the token datatype to the values (and and or).

The attributeUDT type is a complex type composed of an at-
tribute, called ofRelation, and a sequence of elements, name, isINResult and rename. The ofRelation attribute represents a relation ID, to which the attribute belongs. The name element denotes the name of the attribute. The isINResult is an optional element that determines whether the attribute is projected in the final result of the query or not. The rename element represents the new name assigned to the attribute in the query.

6.2 The XREAl Model for Manipulations

The XREAl model formalizes manipulation operations with respect to the manipulation operations in the SQL language. A manipulation operation might be an insert, delete or update operation. Figure 12 shows the XML schema of the moperation component, which might consist of one IStatement, DStatement, or UStatement. The IStatement element consists of attributes, IID and ReceiveAt, and a sequence of elements, rname and attributes. The rname element represents the name of the manipulated relation. The attributes element represents the attributes of the inserted tuple and the corresponding value for each attribute. The DStatement element consists of attributes, DID and ReceiveAt, and a sequence of elements, rname and where. The where element is of type predicateUDT. The UStatement element consists of attributes, UID and ReceiveAt, and a sequence of elements, rname, set and where. The where element is of type predicateUDT. The set element is of type simplePredicate and restricted to use an equal operator only. It is assumed that the update statement is to modify only one attribute at a time.

6.3 An Example

It is assumed that a mobile client, whose ID is MC101, issued the query QCL shown in Figure 4. Figure 13 illustrates an overview of the XREAl specification for the query QCL. This specification consists of a query element. The query ID is QID1. There are two relations (cinema_tab and location_tab), whose RIDs are RID01 and RID02 respectively. These relations are joined together using one join predicate, which is RID01.LID = RID02.LID.

Figure 14 illustrates the XREAl specification for the relation, whose ID is RID01. This specification consists of a relation element. The name of the relation is cinema_tab, and its rename is cin. There is a selection operation over the relation, which is RATE > 4. There is also a projection operation that picks the attributes CNAME, HOTLINE and LID. The order of the elements indicates the order of the operations. In the relation whose ID is RID01, the selection operation precedes the projection operation.

7. UptiME: A MANAGEMENT SYSTEM FOR QUERIES AND CACHES

This section presents a proof-of-concept system, called UptiME (Update Notification Made Easy), that utilizes DBSs as a base for managing queries and caches in mobile information systems. New sub-systems must be introduced to DBSs to support the management of mobile databases. Detecting update relevancy and notifying clients by such updates are examples of the new required sub-systems. UptiME utilizes the DRUPE method and the XREAl model to develop an update notification mechanism as a built-in function inside DBSs that provides XML management support and
triggering mechanism. The following sections discusses the conceptual architecture of the UptiME system and the use of the DBSs utilities for supporting the repository of the XREAL specifications and an update notification mechanism.

7.1 A Conceptual Architecture

The UptiME system provides management support for the contextual information concerning clients, queries issued by these clients and manipulation operations that are to be executed by the server. Moreover, update notification is one of the main functionalities of UptiME in order to preserve the consistency of the database. Figure 18 depicts the conceptual architecture of UptiME that consists of two main layers, the Application Layer and DBS Layer.

The main functionality of the Application Layer is to communicate with external entities, such as mobile clients, and to prepare the contextual information, queries and manipulations to be managed by the DBS Layer. The functionality of the Application Layer is provided through three sub-systems, Mobile Client Manager, Query Manager and Manipulation Manager.

Mobile Client Manager is responsible for registering, unregistering a client and formalizing the contextual information of the client using the XREAL model. The main duties of Query Manager include receiving queries from the registered clients, formalizing the query using the XREAL model, registering it, reply to client queries, and unregistering queries. There is a need to register a query if the client is going to cache the query result. As soon as a client decides to delete cached data extracted from specific queries, Query Manager should be informed to unregister these queries from the system. Manipulation Manager takes the responsibilities for formalizing and registering manipulation operations.

7.2 The XREAL Repository

UptiME utilizes the modern DBSs, which provide XML management support such as DB2 [12] and Oracle [15], to provide a repository for storing the XREAL specifications and update notifications. The XREAL repository is based on a relational database schema, in which XML type is supported to store well-formed and validated XML documents.

Figure 19 depicts the database schema of the XREAL repository. The schema consists of four fundamental relations, mclient, query, manipulation, and notification. A manipulation operation might cause notification(s) to be sent to mobile clients issuing queries, whose cached result intersects with data affected by the manipulation operation.
The relations, mcclient and query, consist of a primary key attribute (MCID and QID) and an attribute of XML type (MCINFO and QTree). Each manipulation operation has an identification number and is classified into three types, insert, delete, and update. The attributes MaID and Type store the identification number and the type of an operation. The both attributes represent the primary key of the relation. Manipulation operations are classified also into two status new (N) or tested (T) operations. The Status attribute represents the status of an operation. The time at which the operation is received is to be stored into the received_at attribute. The ISTMT, DSTMT and USTMT attributes are of XML type and store XML documents representing XREAL specification for insert, delete or update operations respectively. The content of the attributes of XML type is to be validated by the XML schema of the XREAL model.

The notification relation consists of the attributes, MCID, QID, (MaID, Type) and detected_at that represents the time at which the notification is detected. The tuples of the notification relation are to be inserted as a result of testing the intersections between cached and modified data, as it is discussed in the following sub-section.

### 7.3 A Trigger-Based Notification Mechanism

UPTIME utilizes the triggering mechanism provided by DBSs to develop an update notification mechanism as a built-in function of DBSs. The update notification mechanism of UPTIME detects the relevancy of updates (manipulation operations) and notifies clients caching data affected by such updates. Figure 20 depicts a flowchart diagram of the update notification mechanism of UPTIME. This mechanism is based on two triggers created over the manipulation and notification tables.

The trigger attached to the manipulation table invokes a Java stored procedure, called JSPDetective, after inserting a new tuple representing a manipulation operation. JSPDetective starts by generating for each new manipulation operation a list of SQL insert statements based on the DRUP/E method, as discussed in Subsection 7.4. The generation process is implemented using XQuery [18] queries executed by the DBS. Generating a non-empty list means that there is a chance of detecting a client, whose cached data is to be affected by the operation. Then, JSPDetective executes the SQL insert statements, which are based on SELECT statement that might return a tuple to be inserted into the notification table. After executing safely the generated SQL insert statements, JSPDetective executes the manipulation operation, which is to be executed also if the generated list is empty. Finally, JSPDetective changes the status of the manipulation operation to be tested (T). All these actions are processed as a part of the transaction of inserting a manipulation operation into the manipulation table. JSPDetective handles these actions as one sub-transaction, all or nothing. For simplicity, the exception handling is ignored in Figure 20.

The trigger attached to the notification table invokes a Java stored procedure, called JSPNotifier, after inserting new tuple(s) by JSPDetective. For each new tuple JSPNotifier sends a SMS to the clients, whose cached data affected by the manipulation operation.

### 7.4 SQL Templates for the Insert Statements

In case a manipulation operation modifies a specific table, which is used in several queries CQi, there is a probability that several rows are to be added to the notification table. These rows are to be added if and only if there is an intersection between the result of CQi and the modified data. This intersection is to be determined using a SELECT statement, called TEST, returning the number of picked rows. This TEST statement is based on the DRUP/E method.

---

**Figure 19:** The ER diagram of the XREAL Repository

**Figure 20:** A flowchart of the UPTIME notification mechanism.

**Figure 21:** A generic SQL template for the insert statements.

**Figure 22:** An XQuery script for generating the SQL insert statements.

---

The structure of the generated insert statement consists of two dynamic parts. The first part is the row that probably will be added
if and only if the TEST statement returns a positive result. This row is added using SELECT statement over a dummy table. The SELECT statement includes a where clause, which consists of one predicate. This predicate checks that zero is less than the number of rows returned from a specific TEST statement generated dynamically according to the relevant DRU/PE test. Intuitively, the second part is the TEST statement.

Figure 24: A) a SQL template for the insert, B) an example.

Figure 25: A) a SQL template for the delete, B) an example.

Figure 26: A) SQL template for an update operation modifying a projected attribute, B) an example.

from the tables used by th query, and a where clause constructed by adding conjunctively the where clause of the delete operation to the selection and join predicates of the query. Figure 25.B shows the TEST statement for checking the relevancy of the delete operation shown in Figure 16 on the query QCL shown in Figures 13 and 14. As shown in Figure 25.B, the where clause (CID = 9902) of the delete operation is added to the selection and join predicates of the QCL query.

The SQL template of a TEST statement for an update operation modifying a projected attribute is shown in Figure 26.A. The template consists of a SQL selecting from the tables used by th query, and a where clause constructed by adding conjunctively the where clause of the update operation and the negation of the set clause to the selection and join predicates of the query. Figure 26.B shows the TEST statement for checking the relevancy of the update operation number MO3 shown in Figure 5 on the query QCL. As shown in Figure 26.B, the where clause (CID = 9902) and the negation of the set clause (not (hotline = '0721-2059-333')) of the update operation are added to the selection and join predicates of the QCL query.

Figure 26.A depicts a SQL template for an update operation modifying an attribute used in a selection predicate. The TEST statement for the update operation consists of a SQL selecting from the tables used by th query. The where clause of the TEST statement consists of the where clause of the update operation and the join and selection predicates except the selection predicate(s) over the modified attributes, and disjunction predicate that test case D1 and case D3. Figure 26.B shows the TEST statement for checking the relevancy of the update operation shown in Figure 17 on the query QCL. As shown in Figure 27.B, the predicate, ( (RATE > 4) and not (7 > 4) ), checks whether the rate of the cinema was greater than 4 and will
A | B
---|---
select count(*) from the relations of the query where ( ( the where clause of the update operation ) and ( ( the join and selection predicates of the query except the predicates over updated attribute ) or ( ( the selection predicate for D1 ) ) ))) | select count(*) from the relations of the query where ( ( RENEWED_ON = 1999 ) and ( CTAB.LID = LTAB.LID and Postal_Code = '76131' ) and ( ( RATE > 4 ) and ( not(( 7 ) > 4) ) or ( ( not(RATE > 4) and ( ( 7 ) > 4) ) ) )))

Figure 27: A) SQL template for an update operation modifying a selection attribute and B) an example.

not remain greater than 4 after the update or not, case D1. However the predicate, not(RATE > 4) and ( ( 7 ) > 4) checks whether the rate of the cinema was not greater than 4 and will be greater than 4 after the update or not, case D3.

A | B
---|---
select count(*) from the relations of the query where ( ( the where clause of the update statement) and ( ( --Case D1 XC.ID in ( IDs of rows picked by the query) and not NewValue in ( the values used in the join of the query ) ) ) OR ( ( -- CASE D3 ( the selection predicates related to the updated relation) and not XC.ID in ( IDs of rows picked by the query) and NewValue in in ( the values used in the join of the query ) ))) | select count(*) from cinema_tab XC where ( ( RENEWED_ON < 2000) and ( ( --Case D1 XC.CID in ( select the IDs of QCL) and not 101 in (selected values) ) OR ( ( -- CASE D3 ( RATE > 4 ) and not XC.CID in ( select the IDs of QCL) and 101 in (selected values) ) )))

Figure 28: A) SQL template for an update operation modifying a join attribute, B) an example.

Figure 28.A depicts a SQL template for an update operation modifying an attribute used in a join predicate. The TEST statement checks that the updated rows are fall in the intersection D1 or D3. A row is to be part of D1 if and only if the row was part of the query result and the new value is not one of the joining values. A row is to be part of D3 if and only if the row satisfying the selection predicates related the updated table was not part of the query result and the new value is one of the joining values. Figure 28.B shows the TEST statement for checking the relevancy of the update operation number MO5 shown in Figure 5 on QCL. As shown in Figure 28.B, for case D1 the XC.CID should be one of the rows picked in the query result and the value 101 is not one of the joining values used in the query, and for case D3, the updated rows satisfying the selection predicate ( RATE > 4), were not part of the query result and the new value is one of the joining values.

8. EVALUATION

We have utilized DB2 Express-C 9.5 [12] and the Sun Java 1.6 language to implement the DBS Layer shown in Figure 18. The XML Schema of the XReA1 model is used to validate the attributes of XML type shown in Figure 19. The shown SQL and XQuery queries are formalized to be executed using DB2. However, these queries could be supported by other DBSs, which provide XML management support and triggering mechanism. The Application Layer of UPTIME is in-progress. Currently, the developed functionality of the Application Layer is restricted to register mobile clients and queries and to insert manipulation operations. All test were done on a standard PC running Ubuntu 8.04 Linux (Intel(R) Core(TM)2 Duo CPU @ 2.20GHz with 2 GB of RAM). Figure 29 illustrates the time consumption for registering queries on the server and for checking the relevance of insert, update and delete operations. We used the example queries and modifications presented throughout the paper for our experiments.

Figure 29: Evaluation of time consumption

The method of detecting the intersection(s) between modified data and cached data demands a registration of queries issued by mobile clients. These queries are to be stored in the server as individual entries on the XReA1 repository. This method avoids query indexing as needed in other approaches like those presented in [8]. The time that is required to register or de-register a query is the time required to insert or delete an individual query without a need to re-construct the index of the queries. Registering a 16,384 queries took only approximately 6 seconds in our experiments. As shown in Figure 29, the time for registering queries is linear to the number of queries. In other approaches that are based on query indexing, such as presented in [8], the registration time exponentially increases with the growth in the number of queries. Avoiding the query indexing provides high scalability in handling a great number of clients and queries.

Beside the scalability, another advantage of the method presented in this paper is the applicability of the method to be implemented

3The algorithms presented in [8] needed more than 80 seconds for registering 10,000 queries, some others presented in [9] where even slower. However, these results base on a 1.6 GHz Athlon machine with only 512 MB of memory.
within the DBS. That leads to reduce the code complexity of the Application Layer. Consequently, the maintenance of the UptimeME system needs less effort.

The main drawback of this method is the cost of repeating the check for similar queries. However, Figure 29 illustrates the worst case where all registered queries are effected by the issued modification operations. As shown in Figure 29, the time required to check the relevance of a manipulation operation is linear to the number of queries, which use the manipulated table and might be affected by the operation. Our experimental results show that the maximum required time for checking the relevancy of a manipulation operation to 16,384 related queries is approximately 50 seconds. So, we expect, much better performance for real world applications. Furthermore, that drawback of multiple checks for similar queries could be avoided by maintaining a list of similar queries. These experiments with various query loads are part of future work.

9. CONCLUSION AND OUTLOOK

In this paper, we have presented a method called, DRUPE, for detecting relevant updates to cached data. The paper has presented three categories of relevancy test, for insert, delete and update operations respectively. The main idea of these tests is to check the intersection between the modified data and the cached data, which is a result of specific queries. For each manipulation operation, the paper has discussed the effect of the operation on the cached data and the criteria of the intersection(s) between the cached data and modified data. A non-empty intersection means that the manipulation operations are relevant to cached data. This method retrieves the data of the intersection using a query(ies) constructed from a modified data. A non-empty intersection means that the manipulation operations are relevant to cached data. This method retrieves the data of the intersection using a query(ies) constructed from a manipulated table and the queries, whose result is cached on at least one client.

This paper has presented XReAl, an XML-based model, for the queries issued by mobile clients and server-side updates. The paper furthermore has presented a proof-of-concept system, called UptiME that utilizes the DRUPE method and XReAl model to provide a DBS built-in function for update notifications.

The main advantages of our approach to detecting relevant updates to cached data are: 1) the quick response in detecting the relevant updates as soon as the execution of an manipulation operations occurs, 2) the ability to check the intersection between the modified and cached data using SQL queries generated by XQueries, which are executed by an XQuery engine provided within modern DBSs, 3) the flexibility in exchanging and sharing the XReAl specification, such as mobile queries, 4) seamless integration of the update notification management into relational DBSs, and 5) the scalability and performance improvement due to avoiding several intermediate layers that were required to support the management at the application layer.

This paper has presented a research work that is part of a continuous research project aiming at developing a framework for advanced query management in mobile information systems based on XML and DBS utilities. Currently we are extending the UptiME system to support context-aware queries and do additional experiments with different workloads and query sets.

10. REFERENCES