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 Guaranteeing consistency at least demands to invalidate outdated caches.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 XREAL as well as the relevancy proof-of-concept system UPTIME. The main contribution of our work is that the system utilizes the DBS utilities to detect update relevance, 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 invali-

date 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 (**D**etecting **R**elevant **Up**date **E**asily) 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 introduces an XML-based model XREAL (**X**ML-Based **Re**lational **Al**gebra) 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 (**Up**date **N**o**ti**fication **M**ade **E**asy) that utilizes the DRUPE method and the XREAL 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 XREAL 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][1]. 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 QCP complexity but these approaches lead to stricter restrictions to the query language.

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 ta-

[1]based on [14]

bles 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.
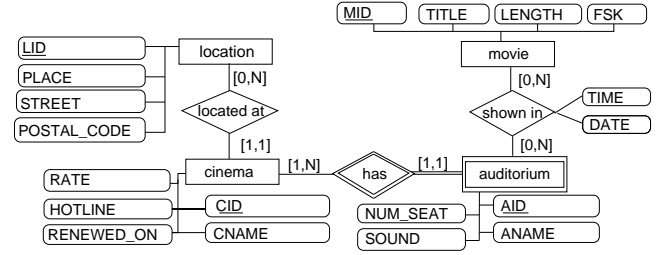

Figure 1: The ER diagram of the cinema database.

The cinema table

CID	CNAME	LID	HOTLINE	RATE	RENEWED_ON
9901	Cineplex	101	111999777	5	1999
9902	Filmpalast	102	111888777	6	2000
9903	City-Kinos	103	111333777	7	1999
9904	ZiZO	101	111555777	2	1999

The location table

LID	PLACE	STREET	POSTAL_CODE
101	Bruchsal	Bahnhofstr	76646
102	Karlsruhe	Brauerstr	76131
103	Karlsruhe	Kaiserstr	76131

Figure 2: Example extensions of some cinema database tables

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.
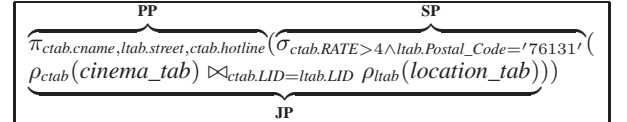
$$\overbrace{\pi_{ctab.cname,ltab.street,ctab.hotline}}^{PP}(\overbrace{\sigma_{ctab.RATE>4 \wedge ltab.Postal\_Code='76131'}}^{SP}(\underbrace{\rho_{ctab}(cinema\_tab) \bowtie_{ctab.LID=ltab.LID} \rho_{ltab}(location\_tab)}_{JP}))$$

Figure 3: The relational algebra of *QCL*

The query notation used in this paper is the notation of the relational algebra operators, such as selection (σ), join ($\bowtie$), projection (π) and rename (ρ). 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*.

- 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.

$$(\pi_{ctab.cname,ctab.hotline,ctab.LID}(\sigma_{ctab.RATE>4}(\rho_{ctab}(cinema\_tab))))$$
$$\bowtie_{ctab.LID=ltab.LID}$$
$$(\pi_{ltab.street,ltab.LID}(\sigma_{ltab.Postal\_Code='76131'}(\rho_{ltab}(location\_tab))))$$

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. Consequentially, we can easily utilize our method with several mobile information systems, in which relational data is queried by mobile clients.

MO1	**insert into** cinema_tab(cid,cname,lid,hotline,rate,renewed_on) **values**(9905, 'Cineplex',102,'111333888',7,2004);
MO2	**delete from** cinema_tab **where** CID = 9903
MO3	**update** cinema_tab **set** hotline = '0721-2059-333' **where** cid = 9902
MO4	**update** cinema_tab set RATE = 7 **where** renewed_on = 1999
MO5	**update** cinema_tab set LID = 101 **where** renewed_on < 2000

Figure 5: Manipulation operations over *cinema_tab*

In the example scenario it is assumed that the server is to execute several manipulation operations over *cinema_tab*. The first operation inserts a new cinema, whose id, name, hotline, rate are 9905, Cineplex, 111333888 and seven, respectively. This cinema is located in Karlsruhe and was renewed in 2004. The second operation deletes the cinema tuple, whose id is 9903. The third operation modifies the hotline of the cinema tuples, whose id is 9902, to 0721-2059-333. The fourth operation updates the rate of the cinema tuples, which were renewed on 1999, to seven. The fifth operation relocates the cinemas, which renewed before 2000, to a location with the id 101. Figure 5 depicts the SQL DDL statements corresponding to these manipulation operations.

5. DRUpE: A METHOD FOR CHECKING UPDATE RELEVANCY

The DRUPE [2] 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.

---
[2] the acronym stands for **D**etecting **R**elevant **Up**date **E**asily

5.1 Test the relevance of inserts

We use the SQL insert construct `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.
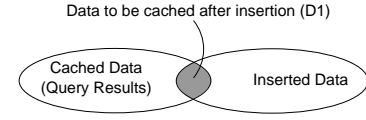

Figure 6: The effect of an insert operation on cached data.

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. Consequentially, the mobile clients should be notified to update their cached data. The intersection is to be not 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 (*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, *MO1*, and a query whose result is cached on at least one mobile client. For example, the intersection $D1$ of the data inserted by *MO1* and the cached data produced by the query **QCL** could be retrieved by the query **QIns**:

$$\sigma_{7>4\wedge ltab.Postal\_Code='76131'\wedge 102=ltab.LID}(\rho_{ltab}(location\_tab))$$

The selection predicate in **QIns** is constructed as follows:

- $7 > 4$ results from the selection predicate of the query **QCL** by replacing the attribute *RATE* with its corresponding value in the inserted tuple of the operation number **MO1** shown in Figure 5.

- $102 = ltab.LID$ is constructed from the join predicate of **QCL** by replacing the attribute *ctab.LID* with its corresponding value in the inserted tuple of the operation number **MO1** shown in Figure 5.

- $ltab.Postal\_Code ='76131'$ is the rest of the selection predicate of **QCL** and associated with the un-manipulated relation(s), in this case *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 *NSP1* by replacing the attributes with their corresponding value in the insert statement, then map the join predicates of *CQ* into selection predicates *NSP2* by replacing the attributes with their values in the insert statement.

---
[3] This only holds for the multiset semantics of the relational algebra.

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.
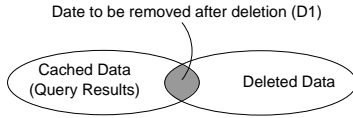

Figure 7: The effect of a delete operation on cached data.

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. Consequentially, 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**:

$$\underbrace{\sigma_{ctab.CID\,=\,9903}}_{\textbf{P1}} \wedge \underbrace{ctab.RATE > 4 \wedge ctab.Postal\_Code =' 76131'}_{\textbf{P2}}$$
$$(\rho_{ctab}(cinema\_tab) \bowtie_{ctab.LID=\,ltab.LID} \rho_{ltab}(location\_tab))$$

The query *QD* consists of the selection predicates: *P1*) and *P2*), and a join predicate $\bowtie_{ctab.LID\,=\,ltab.LID}$ 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* do

    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: `UPDATE relation SET columnName = value [, columnName = value ...] [WHERE condition]`.
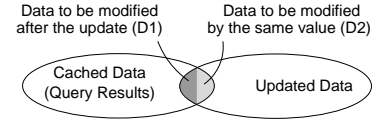

Figure 8: The effect of updating a projected attribute.

The update statement might update attributes belonging to selection, join or projection predicates of queries, whose result is cached on at least one mobile client. If the update statement modifies the value of attributes belonging to projection predicates only, that means no new data will be considered as a part of a query result nor no part of the cached data will be removed. However, part of the cached data might be modified.

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. Consequentially, 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**:

$$\sigma_{((ctab.cid=9902 \land \neg(hotline='0721-2059-333'))}$$
$$_{\land ctab.RATE>4 \land(ctab.Postal\_Code='76131'))}$$
$$(\rho_{ctab}(cinema\_tab) \bowtie_{ctab.LID=ltab.LID} \rho_{ltab}(location\_tab))$$

QU is produced by adding the selection predicates (*ctab.cid* = 9902 $\land \neg$(*hotline* =' 0721 − 2059 − 333')) to **QCL**, such that: *ctab.cid* = 9902 is the WHERE clause of the update statement, and $\neg$(*hotline* =' 0721 − 2059 − 333') 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.
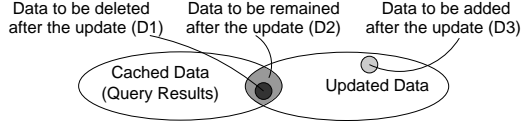

Figure 9: The effect of updating a selection or join attribute.

There are three categories of intersection in the case that the update statement modifies attributes belonging to selection or join predicates. Figure 9 shows these categories, which are:

- Category (*D1*): Data to be deleted after the update , because changing attributes used in selection or join predicates might lead to un-matching.

- Category (*D2*): Data to be remained after the update, although the attributes used in selection or join predicates have been changed, they are still matching the selection and join predicates.

- Category (*D3*): Data to be inserted after the update, the change leads to matching this data with the selection and join predicates, so this data is to be considered as a part of the query result.

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 **QCL** 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 **QCL** could be retrieved by the query **QU2**:

$$\sigma_{((ctab.renewed\_on=1999) \land(((ctab.RATE>4) \land \neg(7>4))}$$
$$_{\lor(\neg(ctab.RATE>4) \land(7>4))) \land(ctab.RATE>4 \land ctab.Postal\_Code='76131'))}$$
$$(\rho_{ctab}(cinema\_tab) \bowtie_{ctab.LID=ltab.LID} \rho_{ltab}(location\_tab))$$

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

- (*ctab.renewed_on* = 1999) is the WHERE clause of the update statement.

- ((*ctab.RATE* > 4) $\land \neg$(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) $\land$ (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 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 **Q** could be retrieved by the query **QU3**:

$$OQ \leftarrow \sigma_{ctab.RATE>4 \land ctab.Postal\_Code='76131'}(\rho_{ctab}(cinema\_tab)$$
$$\bowtie_{ctab.LID=ltab.LID} \rho_{ltab}(location\_tab))$$
$$IDV \leftarrow \pi_{CID,cinema\_tab.LID}(OQ)$$
$$IDN \leftarrow (\sigma_{renewed\_on<2000}(\rho_{ctab}(cinema\_tab))$$
$$\bowtie_{ctab.CID=IDV.CID}(IDV)$$
$$VN \leftarrow \sigma_{IDV.LID\neq101}(IDV)$$
$$D1 \leftarrow \pi_{IDN.CID,IDN.LID}(IDN) \bigcap^{all} \pi_{VN.CID,VN.LID}(VN)$$

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**:

$$NIN \leftarrow \pi_{CID,cinema\_tab.LID}(cinema\_tab) - IDV$$
$$CRW \leftarrow \sigma_{(renewed\_on<2000) \land(RATE>4)}(cinema\_tab)$$
$$VA \leftarrow \sigma_{IDV.LID=101}(IDV)$$
$$WN \leftarrow CRW \bowtie_{CRW.CID=NIN.CID} NIN$$
$$D3 \leftarrow \pi_{WN.CID,WN.LID}(WN) \bigcap^{all} \pi_{VA.CID,VA.LID}(VA)$$

NIN contains the rows that were not picked in the query result. *CRW* contains the updated rows that at the sametime 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 *RATE* > 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 sametime 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**:

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 [4] 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 *moperation*. 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 notiying mobile clients as a DBS built-in function. This mechanism generates the test queries formalized by the DRUPE 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 a *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
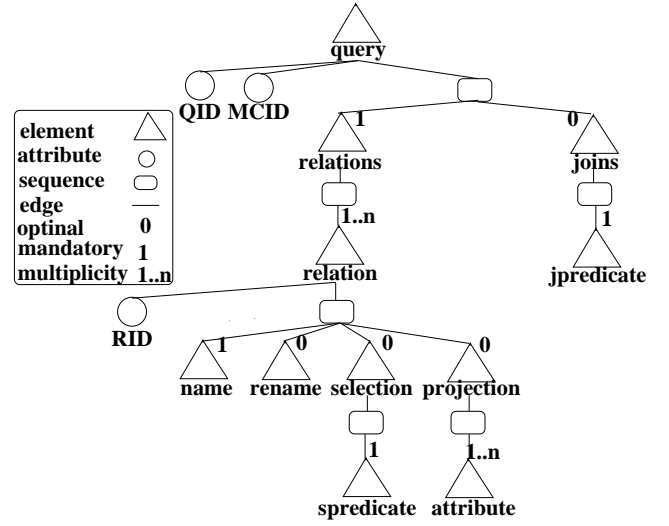
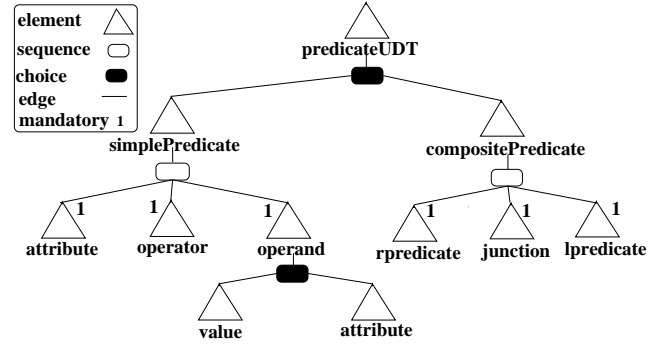**Figure 10: The XML Schema of a relational algebra query.**

Figure 11: The XML Schema of the predicateUDT type.

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, lt, lteq, 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*, *isIN-Result* 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.
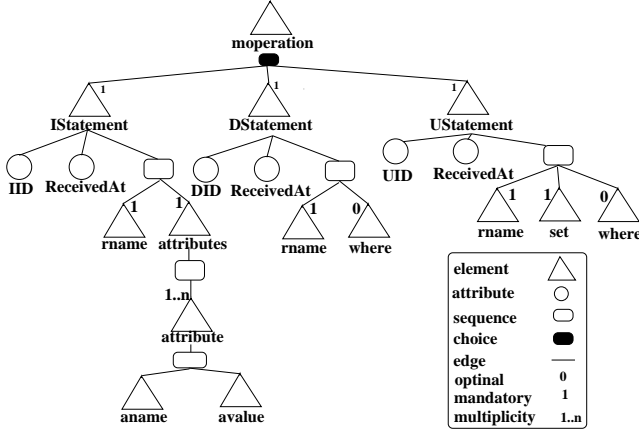

Figure 12: The XML Schema of the manipulation operations.

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 consists 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 *RID*s 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 *ctab*. 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.
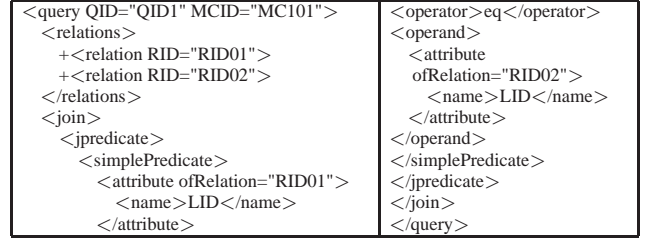

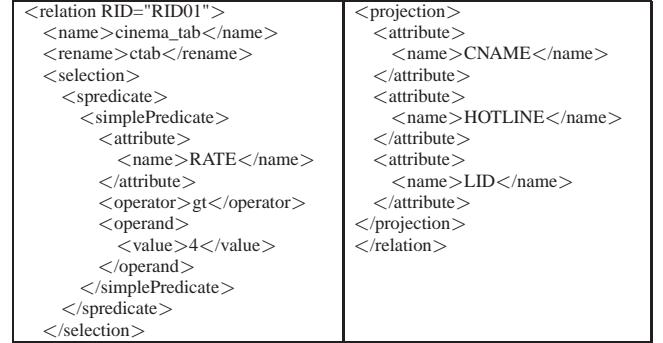**Figure 13: The XREAL specification of the *QCL* query.**

Figure 14: The XREAL specification of the relation RID01.

The operations over the relations, whose IDs are *RID01* and *RID02*, precede the join operation.

Figure 15 illustrates the XREAL specification for the insert operation shown in Figure 5. *IStatement* of the insert operation consists of attributes, *IID* whose value is *I3001* and *receivedAt* that determines the receipt time. There are six elements of type *attribute* that specify the name and value of an attribute, such as *CID* and *9905* for the first attribute of the insert statement.

Figure 16 illustrates the XREAL specification for the delete operation shown in Figure 5. *DStatement* of the delete operation consists of attributes, *DID* whose value is *D5001* and *receivedAt* that determines the receipt time. There is a *where* element under *DStatement* that formalizes the where clause of the delete statement, which is *CID = 9903*.

Figure 17 illustrates the XREAL specification for the update operation shown in Figure 5. *UStatement* of the update operation consists of attributes, *UID* whose value is *U7001* and *receivedAt* that determines the receipt time. The *set* element formalizes the set clause of the update statement, which is *RATE = 7*. The *where* element of *UStatement* formalizes the where clause of the update operation, which is *RENEWED_ON = 1999*.

7. UptiME: A MANAGEMENT SYSTEM FOR QUERIES AND CACHES

This section presents a proof-of-concept system, called UPTIME (**Up**date No**ti**fication **M**ade **E**asy), 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

```
<IStatement IID="I3001"
receivedAt="2008-09-12T11:34:27">
  <rname>cinema_tab</rname>
  <attributes>
    <attribute>
      <aname>CID</aname>
      <avalue>9905</avalue>
    </attribute>
    <attribute>
      <aname>CNAME</aname>
      <avalue>Cineplex</avalue>
    </attribute>
    <attribute>
      <aname>LID</aname>
      <avalue>102</avalue>
    </attribute>
```
```
    <attribute>
      <aname>HOTLINE</aname>
      <avalue>111333888</avalue>
    </attribute>
    <attribute>
      <aname>RATE</aname>
      <avalue>7</avalue>
    </attribute>
    <attribute>
      <aname>
      RENEWED_ON</aname>
      <avalue>2004</avalue>
    </attribute>
  </attributes>
</IStatement>
```

Figure 15: The XREAL specification of the operation *MO1*.

```
<DStatement DID="D5001" receivedAt="2008-09-12T11:34:27">
  <rname>cinema_tab</rname>
  <where>
    <spredicate>
      <simplePredicate>
        <attribute>
          <name>CID</name>
        </attribute>
        <operator>eq</operator>
        <operand>
          <value>9903</value>
        </operand>
      </simplePredicate>
    </spredicate>
  </where>
</DStatement>
```

Figure 16: The XREAL specification of the operation *MO2*.

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 functionality 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* are 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.

```
<UStatement UID="U7001" receivedAt="2008-09-12T11:34:27">
  <rname>cinema_tab</rname>
  <set>
    <spredicate>
      <simplePredicate>
        <attribute>
          <name>RATE</name>
        </attribute>
        <operator>eq</operator>
        <operand>
          <value>7</value>
        </operand>
      </simplePredicate>
    </spredicate>
  </set>
  <where>
    <spredicate>
      <simplePredicate>
        <attribute>
          <name>RENEWED_ON</name>
        </attribute>
        <operator>eq</operator>
        <operand>
          <value>1999</value>
        </operand>
      </simplePredicate>
    </spredicate>
  </where>
</UStatement>
```
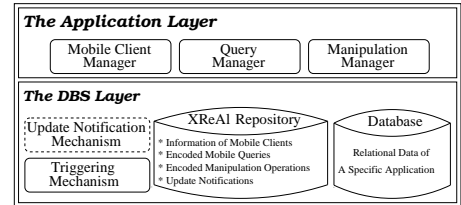
Figure 17: The XREAL specification of the operation *MO4*.

Figure 18: The conceptual architecture of the UptiME System.

The XREAL model is used to encode manipulation operations to be stored into the XREAL repository.

The *DBS Layer* mainly supports the management of the XREAL repository, detecting and notifying clients, whose cached data is affected by manipulation operations. The XREAL repository stores XREAL specifications, such as contextual information, queries and manipulations, and notifications that should be sent to specific mobile clients. Furthermore, the database of a specific application domain, such as the cinema database, is to be managed within the *DBS Layer*. UPTIME extended DBSs, which provide XML data management and triggering mechanism, to react as a mobile database system.

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.
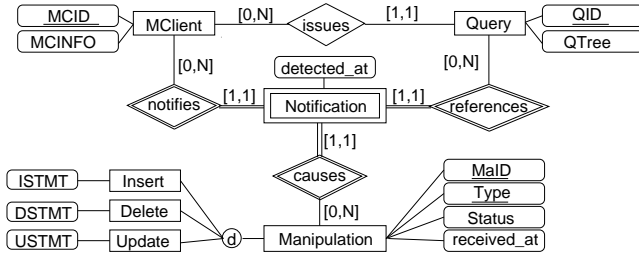
Figure 19: The ER diagram of the XReAl Repository

The relations, *mclient* 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 upadtes (manipulation operations) and notifies clients caching data affected by such updates. Figure 20 depicts a flowchart diagram of the update notification mechanism of UP-TIME . This mechanism is based on two triggers created over the *manipulation* and *notification* tables.
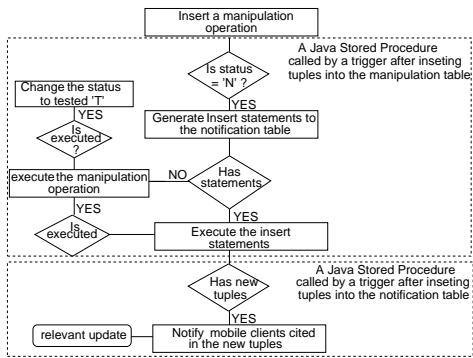

Figure 20: A flowchart of the UPTIME notification mechanism.

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 in-

sert statements based on the DRUPE method, as discussed in Sub-section 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 a 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 DRUPE method.

```
Insert into NOTIFICATION
select MCID?, QID?, MaID?, Type? , current timestamp
from sysibm.sysdummy1
where 0 <
( a SQL query that
    is generated according to the DRUPE method
    and returns the number of picted rows using count(*) )
```

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

```
01 XQUERY
02 <ListOFInsertions>
03 {
04 for $MSTMT in
05 db2-fn:sqlquery("SELECT IT.ISTMT FROM XREAL_INSERTIONS_TAB
06      AS IT WHERE status = 'N'")//IStatement
07 return
08   <Insertions caused_by_MaID="{$MSTMT//@IID}">
09   {
10   for $query in
11   db2-fn:xmlcolumn('QUERY.QTREE')
12     //query[//relation/name="{$MSTMT//rname/text()}"]
13     return
14     <InsertionDDL>
15     Insert into NOTIFICATION
16     select '{$query/data(@MCID)}', '{$query/data(@QID)}',
17       '{$MSTMT//data(@IID)}', 'I' , current timestamp
18     from sysibm.sysdummy1
19     where 0 <
20     (
21     ....
22     ) </InsertionDDL>
23   } </Insertions>
24 } </ListOFInsertions>
```

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 DRUᴘᴇ test. Intuitively, The second part is the *TEST* statement.

```
<ListOFInsertions>
  <Insertions caused_by_MaID="XXXX">
    +<InsertionDDL>
    +<InsertionDDL>
    +<InsertionDDL>
  </Insertions>
</ListOFInsertions>
```

Figure 23: The result of the XQuery.

A	B
select count(*) **from** the relations of the query except the modified relation **where** (The selection and join predicates according to DRUᴘᴇ method for testing an insert operation)	select count(*) **from** location_tab as LTAB **where** (102 = LTAB.LID and 7 > 4 and Postal_Code = '76131')

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

Figure 21 depicts the general structure of the insert statement used to detect clients, whose cached data affected by a manipulation operation. The SELECT statement shown in Figure 21 returns specific values for the client ID, query ID, manipulation ID and type, and the time at which this row will be added to the notification table. SYSIBM.SYSDUMMY1 is a dummy table provided by DB2 to be used for SQL statements, in which a table reference is required but the contents of the table are not important.

Figure 22 shows the query formalized using XQuery to generate an insert statement for each manipulation operation, whose status is new (*N*), and queries, which use the modified table. Lines 15-19 in Figure 22 shows the generation of the first dynamic part of the insert statement. XREAL_INSERTIONS_TAB is the physical table representing the manipulation relation shown in Figure 19 for the insert operation. The generic result of this XQuery is illustrated in Figure 23. The result is an XML document that consists of a list of insert statements caused by a speific manipulation operation. *JSPDetective* after executing the XQuery executes these insert statements.

Figure 24.A depicts a SQL template for an insert operation. The *TEST* statement for the insert operation consists of a SQL selecting from the tables used by the query except the modified query. The where clause of the *TEST* statement constructed from the selection and join predicates of the query except that the selection and join predicate(s) related to the modified table re-constructed using the actual inserted values. Figure 24.B shows the *TEST* statement for checking the relevancy of the insert operation shown in Figure 15 on the query *QCL* shown in Figures 13 and 14. As shwon in Figure 24.B, the selection predicate *RATE > 4* and join predicate *CTAB.LID = LTAB.IID* are both re-constructed by replacing the attributes RATE and CTAB.LID respectively by the corresponding values (7, 102) from the insert statement.

The SQL template of a *TEST* statement for the delete operation is shwon in Figure 25.A. The template consists of a SQL selecting

A	B
select count(*) **from** the relations of the query **where** ((the where clause of the delete) **and** (the selection and join predicates)	select count(*) **from** cinema_tab as CTAB, location_tab as LTAB **where** ((CID = 9903) **and** (CTAB.LID = LTAB.LID and Postal_Code = '76131' and RATE > 4))

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

A	B
select count(*) **from** the relations of the query **where** ((the where clause of the update operation **and** the negation of the set clause of the update operation) **and** (the selection and join predicates of the query))	select count(*) **from** the relations of the query **where** ((cid = 9902 **and** not (hotline = '0721-2059-333')) **and** (CTAB.LID = LTAB.LID and Postal_Code = '76131' and RATE > 4))

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 shwon in Figure 25.B, the where clause (*CID = 9903*) 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 shwon 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 shwon 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 the query. The where clause of the *TEST* statement consists of the where clause of the update, 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 shwon 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
  (
    join and selection predicates
    of the query except the predicates
    over updated attribute
)
  and
  (
    (
    the selection predicate for D1)
    or
    (the selection predicate for D3)
)))
	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
    picted 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
    picted 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 shwon 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 XREAL 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 functionally 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.
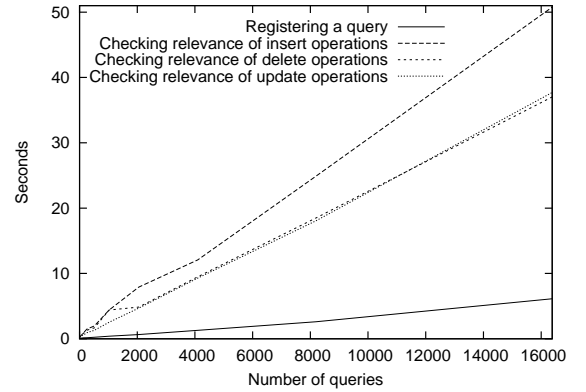

Figure 29: Evaluation of time consumption

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 XREAL 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[5] 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

---

[5]The 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*. Consequentially, the maintenance of the UP-TIME 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 manipulation operation (insert, delete or update) 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

[1] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In A. Gupta and I. S. Mumick, editors, *Materialized Views*, chapter 21, pages 295–322. MIT Press, London, England, 1998.

[2] C. Bunse and H. Höpfner. Resource substitution with components — optimizing energy consumption. In J. Cordeiro, B. Shishkov, A. K. Ranchordas, and M. Helfert, editors, *Proceedings of the 3rd International Conference on Software and Data Technologie*, volume SE/GSDCA/MUSE, pages 28–35, Setúbal, Portugal, July 2008. INSTICC press.

[3] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977.

[4] C. Elkan. Independence of logic database queries and update. In D. J. Rosenkrantz and Y. Sagiv, editors, *Proc. of the ninth ACM symposium on Principles of database systems*, pages 154–160, New York, NY, USA, 1990. ACM Press.

[5] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2007.

[6] H. Höpfner. Towards update relevance checks in a context aware mobile information system. In *Proc. of the 35th annual conference of the German Computer Society (GI)*, volume P-68 of *LNI*, pages 553–557, Bonn, Germany, 2005. Köllen Druck+Verlag GmbH.

[7] H. Höpfner. Update Relevance under the Multiset Semantics of RDBMS. In *Proceedings of the 1. conference on mobility and mobile information systems*, volume P-76 of *LNI*, pages 33–44, Bonn, Germany, 2006. Köllen Druck+Verlag GmbH.

[8] H. Höpfner. Query Based Client Indexing in Client/Server Information Systems. *Journal of Computer Science*, 3(10):773–779, 2007.

[9] H. Höpfner. *Relevanz von Änderungen für Datenbestände mobiler Clients*. VDM Verlag Dr. Müller, Saarbrücken, 2007. in German.

[10] H. Höpfner and C. Bunse. Ressource substitution for the realization of mobile information systems. In J. Filipe, M. Helfert, and B. Shishkov, editors, *Proc. of the 2nd International Conference on Software and Data Technologie*, volume Software Engineering, pages 283–289, Setúbal, Portugal, July 2007. INSTICC press.

[11] H. Höpfner, S. Schosser, and K.-U. Sattler. An Indexing Scheme for Update Notification in Large Mobile Information Systems. In *Current Trends in Database Technology - EDBT 2004 Workshops, Revised Papers*, volume 3268 of *LNCS*, pages 345–354, Berlin, Germany, Nov. 2004. Springer-Verlag.

[12] IBM Redbooks. *DB2 9.5 pureXML Guide*, March, 2008.

[13] D. Maier and J. D. Ullman. Fragments of relations. In M. Stonebraker, editor, *Proc. of the 1983 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1983. ACM Press.

[14] C. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, June 1985.

[15] M. Scardina, B. Chang, and J. Wang. *Oracle Database 10g XML & SQL: Design, Build, & Manage XML Applications in Java, C, C++, & PL/SQL*. McGraw-Hill Osborne Media, 2004. book.

[16] O. Shmueli. Decidability and expressiveness aspects of logic queries. In M. Y. Vardi, editor, *Proc. of the sixth ACM symposium on Principles of database systems*, pages 237–249, New York, NY, USA, 1987. ACM Press.

[17] M. K. Solomon. Some properties of relational expressions. In *Proc. of the 17th annual Southeast Regional Conference*, pages 111–116, New York, NY, USA, 1979. ACM Press.

[18] P. Walmsley. *XQuery*. O'Reilly, first edition edition, March 2007.