Update Relevance under the Multiset Semantics of RDBMS

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

Abstract: In order to reduce transmission costs and response time, mobile clients typically cache data locally. But, avoiding the usage of outdated data and maintaining the consistency of the global database, require the propagation of update information. In this paper we consider the problem of checking the relevance of server side updates for cached data. Therefore, we use test queries performed on a statefull server. At this, the paper discusses issues that especially result from the usage of the multiset semantics of the relational data model.

1 Introduction and Motivation

One major issue of mobile information systems is the price users have to pay for wireless data transmission. Wireless channels are, compared to wired networks, expensive and slow. Therefore, techniques like caching, hoarding or replication are used in order to reduce the costs and, at the same time, to increase the response time of the system. All these techniques create redundant data on the mobile client and "hope" to be able to reuse the local information later on. Redundancy potentially leads to consistency problems. If the local copy of a data item is modified the original has to be updated, too, and vise versa. But, if there are more copies of this data item on other mobile clients, they also have to be updated. Synchronizing updates from one mobile client on a server is a well known and in the literature circumstantially discussed problem. We will not focus this aspect in our paper but discuss a question regarding the other direction. The first issue, one has to consider while trying to update all copies of a data object in a information system with mobile clients, is to determine these mobile clients affected by the update. In other words, the problem is to check the relevance of an update. In [Hö05a] we showed, in the context of a relational database system with mobile clients, that it is not possible to decide about this relevance on the mobile clients themselves if we do not want to restrict the expressiveness of the used query language too much. So we have to use a stateful server. Beside this we showed in [Hö05a], that semantic techniques which try to compare the "update query" to the semantic description of clients contents, underly similar restrictions. To overcome these restrictions we introduced a new approach that uses the data in the database. The semantic description of clients caches is represented in form of a query tree that uses, similar to a Trie [Fr59, Fr60], the syntax of queries. So, queries that use a common prefix of predicates are represented in the same path of the tree. For checking the relevance of an

database update we traverse the tree and compute and execute test queries. Based on the results we decide whether it is necessary to look at the child nodes of the current node. An update is relevant for a client if there is at least one query (path in the tree) that indicates the relevance. In the initial work we considered the relational database system to follow the original, formal definition of relational databases. So, relations were considered to be sets of tuples. However, most (nearly all) relational database management systems support relations under a multiset semantics. That means, that relations may contain duplicates. In this paper we discuss the impacts of this semantics on the test queries and formally deduce these test queries from the relational algebra.

The remainder of the paper is structured as follows. In Section 2 we give a brief overview about related work. Section 3 describes the query representation as well as the format of the allowed update operations. Our approach for checking the relevance of updates under a multiset semantics is included in Section 4. In Section 5 we discuss performance issues by comparing the relevance tests under set semantics to those under multiset semantics. The paper ends with a summary, conclusions and an outlook in Section 6.

2 Related Work

Beside caching, hoarding and replication in mobile information systems, finding irrelevant updates is a task that has to be handled with in the theory of incremental view updates [BCL89]. In fact, a mobile client's data can be considered to be a view over a global database. So far, algorithms were developed that test the relevance or irrelevance by comparing the queries (views) to a query that would result in the updated tuples on a semantic level. There are two major problems with these approaches.

Restrictions to the query language: The algorithms are based on the query containment problem (QCP) [CM77] and suffer from its restrictions. In [So79] it was shown that the QCP is undecidable for arbitrary calculus queries as well as for arbitrary queries in the relational algebra. It is also undecidable for logical query languages [Sh87][1]. But [CM77] includes the proof, that QCP is decidable but NP-complete for conjunctive queries. A lot of 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 [Hö05a]: QCP is defined on the result sets. That means 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. The problem here is, 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 and we would have a containment. We would notify the client about an update that did not change anything.

There are different approaches for handling the incremental view update problem. [MU83] consider inserts and deletes in combination with horizontal database fragments. Therefore,

[1] based on [Pa85]

they do not allow projections. Inserts, deletes and modifications are considered in [BCL89, BCL98]. Here, the restriction is that only equal-joins are supported and self-joins are forbidden. Approaches that use logical query languages typically forbid negations [El90].

Obviously, our approach is also related to cache maintenance, cache invalidation, ad-hoc-replication, and information dissemination in mobile information systems. Because of the given space limitations we do not discuss all these research areas here.

3 Query and Update Representation

The usage of most mobile devices suffer from the limited ergonomics of their input interfaces. Applications on such small and lightweight hardware have to encapsulate the functions for querying the database. Therefore, it is not necessary to support descriptive query languages like SQL. The formal definition of our predicate sequence queries (PSQ) was introduced in [Hö05a]. It is based on the common definition of the relational data model as presented e.g. in [Ma83] which is compatible to Codd's original relational data model [Co69, Co70] but allows to rearrange columns in the result. One problem with both definitions is, that they do not define the name of a relation. We assume - as it is common in database systems - that a relation r over an relation scheme R can be identified by its $\mathtt{name}$ and write such a relation as $r(R_{\mathtt{name}})$. PSQ refers to the relational algebra as well as to some calculus aspects [Co72]. A multiset compatible definition of the relational algebra can be found in [DGK82]. Using the operators of this relational algebra is equivalent to the usage of the original operators. Moreover, existing database management systems internally identify each tuple by using row-ids. So, they can internally use set semantics. However, querying the database is mostly done in multiset semantics[2] because eliminating duplicates is an expensive operation.

PSQ supports conjunctive queries with inequalities and relation renaming. Therefore, self-joins are supported, too. We support three types of predicates:

Selection predicates correspond to the selection operator σ of the relational algebra. Let $\sigma_{\mathsf{sc}}(r(R_{\mathtt{name}}))$ be a selection with the conjunctive selection condition sc which consists of sub-conditions of the form $s = A\ \gamma\ k$ or $s = A\ \gamma\ B$ ($\gamma \in \{\leq, <, =, \neq, >, \geq\}$, $A, B \in R_{\mathtt{name}}$, $k \in dom(A)$). In PSQ this selection can be written as $\{[\mathtt{name}.s_1], \ldots, [\mathtt{name}.s_n]\}$ with $n \in \mathbb{N}$, $n = |\mathsf{sc}|$ and $\forall s_i | 1 \leq i \leq n \rightarrow s_i \in \mathsf{sc}$. SP is the set of all selection predicates.

Join predicates correspond to the θ-join operator of relational algebra. Let $r(R_{\mathtt{name}_1}) \bowtie_{\mathsf{jc}} r(R_{\mathtt{name}_2})$ be a θ-join with the conjunctive join condition jc that consists of sub-conditions of the form $j = A\ \theta\ B$ ($\theta \in \{\leq, <, =, \neq, >, \geq\}$, $A \in R_{\mathtt{name}_1}$, $B \in R_{\mathtt{name}_2}$). In PSQ this join can be written as $[\mathtt{name}_1, \mathtt{name}_2, (j'_1, \ldots, j'_n)]$ with $n \in \mathbb{N}$ and $1 \leq n \leq |\mathsf{jc}|$. Furthermore, we demand a lexicographical order of the relation names within the join predicate. A sub-condition $j_i = A\ \theta\ B$ with $\theta \in \{\leq, <, =, \neq, >, \geq\}$, $A \in R_{\mathtt{name}_1}$ and $B \in R_{\mathtt{name}_2}$ is written as $j'_i = \mathtt{name}_1.A\ \theta\ \mathtt{name}_2.B$. VP is the set[3] of all join predicates.

[2] With SQL the user can force the DBMS to answer with a set by using SELECT DISTINCT.

[3] In order to be consistent with our previous works we use VP instead of JP here.

Projection predicates correspond to the projection operator π of the relational algebra. A projection $\pi_X(r(R_{\texttt{name}}))$ with $X \subseteq R_{\texttt{name}}$ is written as $[\texttt{name}(x_1, \ldots, x_n)]$ with $n \in \mathbb{N}$, $1 \leq n \leq |X|$, and $\{x_1, \ldots, x_n\} = X$. Projections that base on join results $\pi_{X_1, X_2, \ldots, X_i}(r(R_{\texttt{name}_1}) \bowtie_{\theta_1} r(R_{\texttt{name}_2}) \bowtie_{\theta_2} \cdots \bowtie_{\theta_{i-1}} r(R_{\texttt{name}_i}))$ with $i, j \in \mathbb{N}, 1 \leq j \leq i$ and $X_j \subseteq R_{\texttt{name}_j}$, $1 \leq n_j \leq |X_j|$, $\{x_1^j, \ldots, x_{n_j}^j\} = X_j$ are written in PSQ as $[\texttt{name}_1(x_1^1, \ldots, x_{n_1}^1), \texttt{name}_2(x_1^2, \ldots, x_{n_2}^2), \ldots, \texttt{name}_i(x_1^i, \ldots, x_{n_i}^i)]$. PP is the set of all projection predicates.

With $V \subseteq VP$, $pp \in PP \cup \{\varepsilon\}$, $S \subseteq SP$, and $V \cup \{pp\} \cup S \neq \emptyset$ we are now able to formulate a conjunctive query $Q = \bigwedge_{vp \in V} vp \wedge \bigwedge_{sp \in S} sp \wedge pp$ as a predicate sequence query $Q' = \langle vp_1 \ \ldots \ vp_n \ sp_1 \ \ldots \ sp_o \ pp \rangle$ at which

- $\forall i, k \in \mathbb{N}, 1 \leq i < k \leq n; vp_i, vp_k \in V \cup \{\varepsilon\} \Rightarrow vp_i \lhd vp_k$, and
- $\forall i, k \in \mathbb{N}, 1 \leq i < k \leq o; sp_i, sp_k \in S \cup \{\varepsilon\} \Rightarrow sp_i \lhd sp_k$

must hold. Here, $\lhd$ means lexicographically smaller. We have discussed more requirements that guarantee the correctness of PSQ-queries in [Hö05a]. However, it is possible to translate PSQ to SQL in order to perform the query. As mentioned above, we also support the renaming of relations which is an essential issue for self-joins. Therefore, the `name` of a relation is transformed into `name@alias`. This corresponds to the TABLE_NAME-AS-ALIAS-construct of SQL.

In contrast to the queries posted via mobile devices, we assume that update operations on the server are formulated in SQL. Therefore, we have to differ between three possible update operations:

Inserting a tuple: In order to insert a tuple into a relation one can use the following construct: `INSERT [INTO] [[database_name.]owner.] {table_name | view_name} [(column_list)] {[DEFAULT] VALUES | VALUES (value [,...]) | SELECT_statement}` . We use a simplified notation: `INSERT INTO name (column_list) VALUES (value [,...])`. Furthermore, we assume that `column_list` includes all attributes of the relation. Inserting more than one tuple per query, as it is allowed in the standard, is not allowed here. But, one can execute more inserts to realize this issue.

Deleting tuples: Similar to the insertion of tuples we use a simplified SQL statement here: `DELETE FROM name WHERE clause`. We assume that the selection condition `clause` includes only conditions of the form $A \ \phi \ constant$ with the attribute name A and $\phi \in \{<, <=, !=, =, >=, >\}$.

Modifying tuples: Since we use the word "update" in a general manner, we have to use the word "modification" here, even if the corresponding SQL statement begins with UPDATE. The simplified notation of a modification is `UPDATE name SET column_name = expression [,...n] WHERE condition`.

Last but not least, we ignore integrity constraints in the following. This means that we assume a posted update to be performed really.

4 Relevance Tests

As described in Section 1 queries are stored on the server in form of a query tree, and the relevance of an update is checked by traversing the tree. We here ignore the storage structure but deduce a predicate based relevance check. The idea is, that first join predicates are checked. If this test shows the irrelevance, we can stop. Otherwise we have to check the selection predicates and, if necessary, the projection predicate. Of course, an update can only be relevant for the client that posted a query if the update operation affects a relation that is used in the query. If this condition that can be easily checked without accessing the database does not hold, then we can stop. Otherwise we have to start with the relevance tests which are performed *before* the update is performed at all.

4.1 Testing the relevance of inserts

Let $r(R_{\texttt{name}})$ be the relation that is used for inserting the tuple t_i. Furthermore, $A = (a_1^i, \ldots, a_n^i)$ are the attributes of this relation (and also the attributes of the inserted tuple; from `(column_list)`), and $X = (x_1^i, \ldots, x_n^i)$ are the corresponding attribute values (from `(value [,...])`).

Inserting a tuple affects the join result if there is no join condition that removes the tuple from the result. With the knowledge of `name`, A, X, and the formal definition of the relational algebra we can create a join-test by substituting $r(R_{\texttt{name}})$ for $\{t_i\}$ in the join predicates. Thus, we create a virtual relation $\{t_i\}$ and perform the join. The insert can not be relevant for this query if the result TJ of the performed (modified) join is empty. Of course, if a query contains more than one join predicate,we can check all of them together. Let us assume that the tuple will be inserted into relation $r(R_1)$. Let us furthermore assume, that we have the following "join-chain": $r(R_1) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$. Therefore, the insert can only be relevant if $TJ = \{t_i\} \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n) \neq \emptyset$ holds. However, this formula is only correct if the query does not include a self-join ($\forall R_i, R_j | 1 \leq i < j \leq n \wedge r(R_i) \neq r(R_j)$). Having a self-join of the relation $r(R_1)$ we have to consider all possible substitutions[4] of $\{t_i\}$ for $r(R_1)$. In this case one can write the join-chain as:

$$\underbrace{r(R_1^1) \bowtie_{\mathsf{jc}_{S_1}} r(R_1^2) \bowtie_{\mathsf{jc}_{S_2}} \cdots \bowtie_{\mathsf{jc}_{S_m}} r(R_1^{m+1})}_{\text{self-join}} \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$$

Obviously, the inserted tuple would affect all relations $r(R_1^1), r(R_1^2)$, until $r(R_1^{m+1})$. May $vp = [\texttt{name@alias1}, \texttt{name@alias2}, (\Theta)]$ be a join predicate representing a self-join only. An inserted tuple can only be relevant for a query containing vp if $TJ = (r(R_{\texttt{name}}^{\texttt{alias1}}) \bowtie_\Theta \{t_i\}) \cup (\{t_i\} \bowtie_\Theta r(R_{\texttt{name}}^{\texttt{alias2}})) \cup (\{t_i\} \bowtie_\Theta \{t_i\}) \neq \emptyset$ holds.

If $TJ \neq \emptyset$ and the query does not include selection predicates, then the update is relevant because it changes the cardinality of the result, and such an effect can not be reverted

[4] For an m-fold self-join (meaning the relation appears $m+1$ times) we have to consider $2^{m+1}-1$ alternative substitutions.

by a projection. But, selection predicates could remove the new join result tuples (see Figure 1). Therefore, we have to check them. Having the selection predicates $sp_n \in SP$ with $n \in \mathbb{N}, 1 \leq n \leq |SP|$ that use the selection conditions sc_n, the insert is irrelevant if $TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{sc}_m}(TJ) = \emptyset$ holds. If we have a query with selection predicates but without join predicates, then we can use the substitution again. In this case the insert is irrelevant if $TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{sc}_m}(\{t_i\})$ holds.

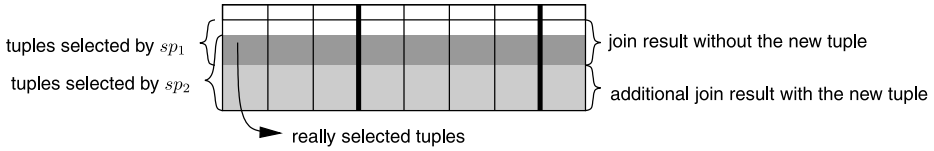

Figure 1: inserted tuple is relevant for join but not for selection

As mentioned above we do not have to check the relevance of an insert regarding a projection predicate because a projection is not able to decrement the cardinality of a relation.

4.2 Testing the relevance of deletes

Let $r(R_{\mathtt{name}})$ be the relation used for deleting tuples. From the WHERE-clause of the given delete statement we can learn the delete condition dc. So we can check, which tuples would be deleted by performing the selection $\sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}))$ first. If the result is the empty set, then we can stop because no data will be deleted. Otherwise, we have to calculate additional test queries.

Considering the (self-join free) join chain $r(R_1) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$ represented by the join predicates of a query, the delete operation is irrelevant if $TJ = \sigma_{\mathsf{dc}}(r(R_1)) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n) = \emptyset$ holds. This means that the deleted tuples does not flow into the result of the join. The test query for self-joins like $r(R_1^1) \bowtie_{\mathsf{jc}_{S_1}} r(R_1^2) \bowtie_{\mathsf{jc}_{S_2}} \cdots \bowtie_{\mathsf{jc}_{S_m}} r(R_1^{m+1}) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$ is a little bit more complicated. Let us first assume, that $vp = [\mathtt{name@alias1}, \mathtt{name@alias2}, (\Theta)]$ is the only join predicate of the query. The idea now is similar to the insert handling but we substitute $r(R_{\mathtt{name}})$ for $\sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}))$. So, the delete is irrelevant if

$$\begin{aligned}
&(r(R_{\mathtt{name}}^{\mathtt{alias1}}) \bowtie_{\Theta} \sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}^{\mathtt{alias2}})))\cup \\
&(\sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_{\Theta} r(R_{\mathtt{name}}^{\mathtt{alias2}}))\cup \\
&(\sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_{\Theta} \sigma_{\mathsf{dc}}(r(R_{\mathtt{name}}^{\mathtt{alias2}}))) = \emptyset
\end{aligned}$$

holds. To optimize this test query we adapt the delete condition dc to the aliases ($\mathsf{dc}^{\mathtt{alias}}$). Obviously this adaption is correct because each alias for a relation can be interpreted as a

independent relation. The result of this optimization is the following test query:

$$\begin{aligned}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta \sigma_{\mathsf{dc}^{\texttt{alias2}}}(r(R_{\text{name}}^{\texttt{alias2}})))\cup\\ (\sigma_{\mathsf{dc}^{\texttt{alias1}}}(r(R_{\text{name}}^{\texttt{alias1}})) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\cup\\ (\sigma_{\mathsf{dc}^{\texttt{alias1}}}(r(R_{\text{name}}^{\texttt{alias1}})) \bowtie_\Theta \sigma_{\mathsf{dc}^{\texttt{alias2}}}(r(R_{\text{name}}^{\texttt{alias2}})))\end{aligned}$$

The next step is to rewrite the test query by using the explicit delete condition naming.

$$\begin{aligned}\sigma_{\mathsf{dc}^{\texttt{alias2}}}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\cup\\ \sigma_{\mathsf{dc}^{\texttt{alias1}}}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\cup\\ \sigma_{\mathsf{dc}^{\texttt{alias1}}\wedge\mathsf{dc}^{\texttt{alias2}}}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\end{aligned}$$

From the database theory, especially from the rule based optimization of queries, we know $\sigma_{pred1}(r(R)) \cup \sigma_{pred2}(r(R)) \equiv \sigma_{pred1\vee pred2}(r(R))$. Therefore, the test query becomes

$$\begin{aligned}\sigma_{\mathsf{dc}^{\texttt{alias2}}\vee\mathsf{dc}^{\texttt{alias1}}\vee(\mathsf{dc}^{\texttt{alias1}}\wedge\mathsf{dc}^{\texttt{alias2}})}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\cup\\ (r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))\cup\\ (r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}})).\end{aligned}$$

Obviously it is possible to improve the query by using simple rules from the logics and from set theory. The result is:

$$\sigma_{\mathsf{dc}^{\texttt{alias2}}\vee\mathsf{dc}^{\texttt{alias1}}}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_\Theta r(R_{\text{name}}^{\texttt{alias2}}))$$

Finally it can be shown, that a delete operation is irrelevant for a join chain including an m-fold self-join if and only if

$$\begin{aligned}TJ = \sigma_{\mathsf{dc}^{\texttt{alias1}}\vee\ldots\mathsf{dc}^{\texttt{alias}m+1}}(r(R_{\text{name}}^{\texttt{alias1}}) \bowtie_{\mathsf{jc}_{S_1}} r(R_{\text{name}}^{\texttt{alias2}}) \cdots \bowtie_{\mathsf{jc}_{S_m}} \ldots\\ \ldots r(R_{\text{name}}^{\texttt{alias}m+1})) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n) = \emptyset.\end{aligned}$$

If a relevance of the update regarding the join predicates was found, then we have to test the selection predicates $sp_n \in SP$ with $n \in \mathbb{N}, 1 \leq n \leq |SP|$. Since deletion decreases the cardinality of the join result, it is not necessary to test the relevance regarding a projection predicate (similar to inserting). Nevertheless, the delete is irrelevant for a query with selection and join predicates if $\sigma_{\bigwedge_{m=1}^{n}\mathsf{sc}_m}(TJ) = \emptyset$ holds (see Figure 2). The test query for queries without join predicates but with selection predicates is $\sigma_{\mathsf{dc}\wedge\bigwedge_{m=1}^{n}\mathsf{sc}_m}(r(R_{\text{name}}))$. Here, sc_m is the select condition of the selection predicate sp_m and, again, the delete is irrelevant if the result of the test query is the empty set.

4.3 Testing the relevance of modifications

The most expensive relevance test is testing modifications. In previous works [HSS04] we considered updates to be combinations of delete and insert operations. In most practical

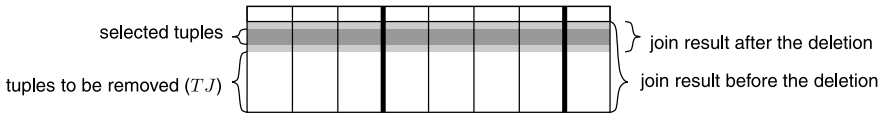

Figure 2: deletion is relevant for the join predicates but not for the selection predicates

cases this assumption is sufficient for the relevance test. But, if we consider theoretical correctness, then we have to handle the case that the modification replaces tuples by themselves. Figure 3 illustrates this case with an easy example. However, there might be more complex cases that have the same result.

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

→`UPDATE SET A=1 WHERE A=1`→

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

Figure 3: modification does not change anything in $r(R_{\texttt{name}})$

If we now consider modifications as delete-insert-combinations, then we could recognize the relevance of the delete and insert without having real changes. Another problem raises from the fact, that we only allow single tuples to be inserted whereas updates are allowed to modify more than one tuple.

From the WHERE-clause of the UPDATE-statement we can learn the condition uc that specifies those tuples that will be modified. Let uexpr be the modification function that can be learned for the UPDATE-statement (`column_name=expression`), too. So, the modification can be suggested as the mapping u : **DAT**$(\mathcal{S})$ → **DAT**$(\mathcal{S})$. At this, **DAT**$(\mathcal{S})$ with the database schema $\mathcal{S} = (S, \Gamma)$ is the set of correct database states. $S = \{r(R_1), \ldots, r(R_p)\}$ is the set of relations and Γ is the set of integrity constraints. As discussed in Section 3 we ignore Γ here and allow only one-relation-updates. So, it is possible to consider updates to be mappings between (multi-)sets of tuples $ut : r(R) \rightarrow r(R)$:

$$ut(t) = \begin{cases} \mathsf{uexpr}(t) & \text{if } \sigma_{\mathsf{uc}}(\{t\}) \neq \emptyset \\ t & \text{otherwise} \end{cases}$$

The resulting mapping $ur : S \rightarrow S$ for updating a relation is $ur(r(R)) = \bigcup_{m=1}^{|r(R)|} ut(t_m)$ with $t_m \in r(R)$.

A modification of relation $r(R_{\texttt{name}})$ can only be relevant if $r(R_{\texttt{name}}) \neq ur(r(R_{\texttt{name}}))$ holds. Without the need to look at the query we can except a relevance if

- no tuples are selected for modification ($\sigma_{\mathsf{uc}}(r(R_{\texttt{name}})) = \emptyset$), or
- if the modification does not change anything ($ur(\sigma_{\mathsf{uc}}(r(R_{\texttt{name}})) = \sigma_{\mathsf{uc}}(r(R_{\texttt{name}}))$).

We only have to take the predicate tests into account if both conditions does not hold. In contrast to insert and delete operations a modification does not have to change the cardinality of a query result (see Figure 4).

$r(R_{\text{name}})$	A	B
	1	X
	1	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\text{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	Z	1	W

```
UPDATE name SET B='Y' WHERE B='Z'
```

$r(R_{\text{name}})$	A	B
	1	X
	1	**Y**

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\text{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	**Y**	1	W

Figure 4: Modification does not change the cardinality

Modifications can change single attribute values. In order to define the resulting test query we first consider the join result as a selection based on the Cartesian product. With the join result after the modification TJ_{new} and the current (old) join result TJ_{old}, the modification is relevant if $TJ = (TJ_{old} \cup TJ_{new}) - (TJ_{old} \cap TJ_{new}) \neq \emptyset$ holds. We also optimized this formula in [Hö05a]. Because of the given space limitations we do not discuss this optimization, that are based on the set theory here, in detail but present the result. The idea is, that TJ_{new} results from TJ_{old}. So, it is possible to write it as $TJ_{new} = (TJ_{old} - J_{bu}) \cup J_{au}$. At this, J_{bu} are the tuples that will be removed from the result and J_{au} are the tuples added to the result. The resulting test query is $TJ = [TJ_{old} \cup J_{au}] - [(TJ_{old} - J_{bu}) \cup (TJ_{old} \cap J_{au})]$. If the PSQ-query includes an m-fold self-join, then the test query is composed using

- $J_{au} = \underbrace{ur(\sigma_{\mathsf{uc}}(r(R_1^1))) \bowtie_{\mathsf{jc}_{S_1}} ur(\sigma_{\mathsf{uc}}(r(R_1^2))) \cdots \bowtie_{\mathsf{jc}_{S_m}} ur(\sigma_{\mathsf{uc}}(r(R_1^{m+1})))}_{\text{self-join}} \cdots \bowtie_{\mathsf{jc}_1} r(R_2) \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$ and
- $(TJ_{old} - J_{bu}) = \sigma_{\mathsf{uc}^{\mathtt{alias1}} \vee \ldots \mathsf{uc}^{\mathtt{aliasm+1}}}(r(R_{\mathtt{name}}^{\mathtt{alias1}}) \bowtie_{\mathsf{jc}_{S_1}} r(R_{\mathtt{name}}^{\mathtt{alias2}}) \cdots \bowtie_{\mathsf{jc}_{S_m}} \ldots r(R_{\mathtt{name}}^{\mathtt{aliasm+1}})) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$.

For PSQ-queries without self-joins the components are

- $J_{au} = ur(\sigma_{\mathsf{uc}}(r(R_1))) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$ and
- $(TJ_{old} - J_{bu}) = \sigma_{\mathsf{uc}}(r(R_1)) \bowtie_{\mathsf{jc}_1} r(R_2) \bowtie_{\mathsf{jc}_2} \cdots \bowtie_{\mathsf{jc}_n} r(R_n)$.

If the modification is irrelevant for join predicates, then we can stop here. Otherwise we have to look at selection predicates (if there are selection predicates) *and* at the projection predicate (if there is one). The modification is relevant for the selection predicates if $TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{sc}_m}(TJ) \neq \emptyset$. The relevance for queries without joins but with selections

can be checked similar to the join predicates. At this, the modification is irrelevant if

$$TS = [\sigma_{\bigwedge_{m=1}^{n} \mathtt{sc}_m}(r(R_{\mathtt{name}})) \cup \sigma_{\bigwedge_{m=1}^{n} \mathtt{sc}_m}(ur(r(R_{\mathtt{name}})))] - [\sigma_{\bigwedge_{m=1}^{n} \mathtt{sc}_m}(r(R_{\mathtt{name}})) \cap \sigma_{\bigwedge_{m=1}^{n} \mathtt{sc}_m}(ur(r(R_{\mathtt{name}})))] = \emptyset$$

holds.

Unfortunately, projections can remove modified data from the result without decreasing the cardinality of the result. For example, the modification in Figure 4 is irrelevant if the query is answered with the attributes A, C, and D only. Thats why, we have to test the relevance of a modification for projection predicates.

The modification is relevant for a query if at least one modified attribute is included in the result. QA is the set of attributes projected by Q. From `column_name=expression` of the modifications we can learn the set of attributes UA. The modification is irrelevant[5] if $QA \cap UA = \emptyset$.

5 Performance Issues

Obviously, compared to the semantic approaches referenced in Section 2 our approach that uses the database has to be slower. Furthermore, we have to point out, that the performance of our approach depends not only on the introduced algorithm with its test queries but also on the capabilities of the underlying database system (DBS). But, reducing the limitations of the usable query language requires the usage of the DBS. In the following we do not present the results of a real world evaluation but compare the performance of the update relevance tests under multiset semantics to the performance of such tests under set semantics.

The first proof of the benefits of multiset semantics is shown in Figure 5. Here we used an emulated multiset semantic and discussed the duration of an insert relevance test. As discussed in Section 4 we do not have to consider projection predicates in multisets. Emulated means here that we performed the same inserts[6] to the same increasing number of queries on the same machine but ignored the projection test. The benefit is apparently.

The major benefit of the multiset semantics is, that we have the guarantee, that an insert or a delete changes the cardinality of the result and a projection can not affect the cardinality, whereas under set semantics projections are able to affect the cardinality. Furthermore, the relevance test for projections in the case of a modification is database independent under multiset semantics whereas it is a real database test query under set semantics (see again [Hö05a]).

Last but not least, the posted test queries does not have to use `SELECT DISTINCT` under multiset semantics. Therefore they, can be performed faster since `DISTINCT` requires duplicate elimination which is based on the sorting of a relation.

[5] Due to the duplicate elimination under set semantics this test would not be sufficient!

[6] For a detailed evaluation of the relevance tests under set semantic see [Hö05a]

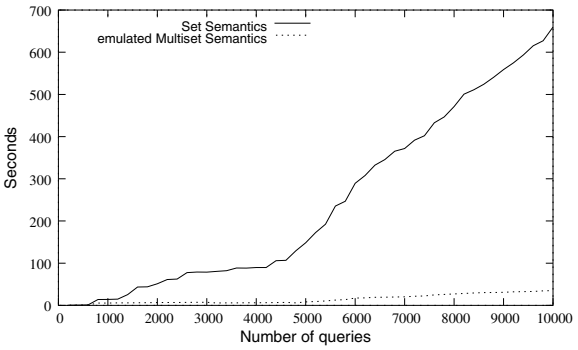

Figure 5: Testing the relevance of an insert under emulated multiset semantics and set semantics

Obviously, the execution of relevance tests under multiset semantics has to be more high-performance than tests under set semantics.

6 Summary, Conclusions and Outlook

In this paper we presented our approach for checking the relevance of server side inserts, deletes and modifications for the caches on mobile database clients. We showed, that comparable approaches suffer from strict restrictions. Furthermore, we deduced the required relevance tests that benefit from the usage of the multiset semantics of relation database systems. To conclude the paper we can point out that the usage of multiset semantics reduces the number of test queries that have to be performed on the database and therefore, increases the performance of our approach.

The next step now is to evaluate the multiset based relevance check in more detail and to combine it with available dissemination approaches. Furthermore, we are working on completing the relational completeness of the supported query language which requires also the set operations union, difference and intersection. The global vision behind our work is to create a context aware information system with mobile clients on the basis of standard database technologies. The first steps are done but now we have to combine the relevance issues with our context model (see [Hö05b] for the first results).

References

[BCL89] Blakeley, J. A., Coburn, N., and Larson, P.-A.: Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems (TODS)*. 14(3):369–400. September 1989. ebenfalls publiziert als [BCL98].

[BCL98] Blakeley, J. A., Coburn, N., and Larson, P.-A.: Updating derived relations: Detecting

irrelevant and autonomously computable updates. In: Gupta, A. and Mumick, I. S. (Eds.), *Materialized Views*. chapter 21, pp. 295–322. MIT Press. London, England. 1998.

[CM77] Chandra, A. K. and Merlin, P. M.: Optimal implementation of conjunctive queries in relational data bases. In: *Proc. of the ninth annual ACM symposium on Theory of computing*. pp. 77–90. New York, NY, USA. 1977.

[Co69] Codd, E. F.: Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. *IBM Research Report, San Jose, California*. RJ599. 1969.

[Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM (CACM)*. 13(6):377–387. 1970.

[Co72] Codd, E. F.: Relational Completeness of Data Base Sublanguages. In: Rustin, R. J. (Ed.), *Data Base Systems (Proceedings of the 6th Courant Computer Science Symposium, May 24-25, 1971, New York, N.Y.)*. Automatic Computation. pp. 65–98. Englewood Cliffs, New Jersey. 1972. Prentice-Hall.

[DGK82] Dayal, U., Goodman, N., and Katz, R. H.: An extended relational algebra with control over duplicate elimination. In: Ullman, J. D. and Aho, A. V. (Eds.), *Proc. of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*. pp. 117–123. New York, NY, USA. 1982. ACM Press.

[El90] Elkan, C.: Independence of logic database queries and update. In: Rosenkrantz, D. J. and Sagiv, Y. (Eds.), *Proc. of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. pp. 154–160. New York, NY, USA. 1990. ACM Press.

[Fr59] Fredkin, E.: *Trie memory*. Information Memorandum, Bolt Beranek and NewMan Inc. Cambridge, MA. 1959.

[Fr60] Fredkin, E.: Trie memory. *Communications of the ACM*. 3(9):490–499. August 1960.

[Hö05a] Höpfner, H.: *Relevanz von Änderungen für Datenbestände mobiler Clients*. Dissertation. Department of Computer Science, University of Magdeburg. January 2005. in German.

[Hö05b] Höpfner, H.: Towards Update Relevance Checks in a Context Aware Mobile Information System. In: *INFORMATIK 2005 - Informatik LIVE! (Band 2)*. volume P-68 of *LNI*. pp. 553–557. Bonn, Germany. 2005. Köllen Druck+Verlag GmbH.

[HSS04] Höpfner, H., Schosser, S., and Sattler, K.-U.: An Indexing Scheme for Update Notification in Large Mobile Information Systems. In: Lindner, W., Mesiti, M., Türker, C., Tzikzikas, Y., and Vakali, A. (Eds.), *Current Trends in Database Technology*. volume 3268 of *LNCS*. pp. 345–354. Berlin. November 2004. Springer-Verlag.

[Ma83] Maier, D.: *The Theory of Relational Databases*. Computer Science Press, Inc. Rockville, Maryland. 1983.

[MU83] Maier, D. and Ullman, J. D.: Fragments of relations. In: Stonebraker, M. (Ed.), *Proc. of the 1983 ACM SIGMOD international conference on Management of data*. New York, NY, USA. 1983. ACM Press.

[Pa85] Papadimitriou, C.: A note on the expressive power of prolog. *Bulletin of the EATCS*. 26:21–23. June 1985.

[Sh87] Shmueli, O.: Decidability and expressiveness aspects of logic queries. In: Vardi, M. Y. (Ed.), *Proc. of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. pp. 237–249. New York, NY, USA. 1987. ACM Press.

[So79] Solomon, M. K.: Some properties of relational expressions. In: *Proc. of the 17th annual Southeast Regional Conference*. pp. 111–116. New York, NY, USA. 1979. ACM Press.