An Indexing Scheme for Update Notification in Large Mobile Information Systems*

Hagen Höpfner[1], Stephan Schosser[1], and Kai-Uwe Sattler[2]

[1] Otto-von-Guericke University of Magdeburg, Department of Computer Science,
P.O.Box 4120, D-39016 Magdeburg, Germany
{hoepfner@iti.cs.uni-magdeburg.de| schosser78@arcor.de}
[2] Technical University of Ilmenau, Department of Computer Science and Automation,
P.O.Box 10 0565, D-98684 Ilmenau, Germany
kus@tu-ilmenau.de

Abstract. Due to the increasing usage of small and low footprinted devices like mobile phones as clients of mobile information systems a new problem arises: "How to determine the relevance of updates for a large number of mobile clients?" In this paper we present an indexing scheme that represents conjunctive queries posed by the mobile clients in a trie. So, IDs of the clients are referenced by their queries and checking the relevance of an update can be efficiently done by a trie lookup.

1 Introduction and Motivation

Most mobile information system are designed as an add-on to existing classical, fixed network based information systems. The mobile clients have to connect to the fixed network via a base station. But, normally such systems do not consider the extremely increasing number of mobile devices that are usable for accessing the data. In fact, mobile phones, smart phones and networked PDAs will be used as information system clients. So, a new central challenge for supporting mobile devices on the server site arises: *How to handle interest of a large number of mobile clients efficiently?*

The light-weightiness of the mobile clients and the classical problem of incrementally updating materialized views prohibit to transfer all updates directly to the mobile clients. So, client queries have to be stored and evaluated on the server. Now, if an update occurs it is obviously inefficient to check all stored queries sequentially.

An alternative, which is presented in this paper, is a query index based update evaluation approach that allows to look up such mobile devices which are potentially interested in updated information. At this, queries are represented as paths in a trie [2] whereby each path references a set of mobile client IDs.

The remainder of the paper is structured as follows. In Section 2 we discuss related work and point out the differences between our work and overlapping research areas. Section 3 describes the query index and how it is used to look up mobile clients efficiently. The evaluation of our approach can be found in Section 4. Finally, the paper closes with conclusions and an outlook on ongoing research in Section 5.

* This research is supported by the DFG under grant SA 782/3-2

W. Lindner et al. (Eds.): EDBT 2004 Workshops, LNCS 3268, pp. 345–354, 2004.

2 Related Work

Our work is embedded into the context of mobile databases and information systems. We do not support completely wireless systems but systems that allow mobile clients to connect via a base station to a fixed network as discussed in [11]. Furthermore, several works regarding the replication and synchronization of data between a static server and mobile clients has been done. But these approaches, that can be classified as data centric (e.g. [9]) and transaction centric (e.g. [3, 10]), consider the integration of offline executed update operations from the mobile clients to the server. However, we currently do not take into account the synchronization of such updates but focus on performance aspects of notifying mobile clients about updates on the server.

Beside this, our research can be interpreted as profile handling. That means, that the stored queries describe the profiles of mobile users as it is done in [1] or [14]. But profiles are based on a semantic based selection of needed data. We plan to support also semantic queries regarding user contexts in our future work.

Another related research area concerns the query containment problem that is considered in many publications, e.g. [4, 12]. [13] comprises the complexity issues of various kinds of queries that are represented as conjunctively connected predicates. However, we have to deal with query containment only when we use semantic information for query indexing. Here, we focus on an approach using syntactical information which could be extended to use semantic information.

Last but not least we have to point out, that there exist relationships between the problems that are focused in our work and the common problem of incrementially updating materialized views [8]. If we consider stored queries as view definitions we have also to decide which "view" is affected by an update. But we do not have to materialize the update on the mobile clients as yet.

3 Indexing Mobile Clients Using a Trie

As already mentioned, a sequential check of all registered queries is inefficient. Therefore, we introduced in [6] first ideas on a trie-based indexing of mobile clients. Database queries are represented as conjunctively connected predicates that are ordered in an alpha-numerical predicate order. We currently support three[1] different kinds of predicates *relation predicates*, *join predicates*, and *selection predicates*. A relation predicate r is comparable to the projection operator of the relational algebra. A projection $\pi_X(r(R))$ with $X \subseteq R$ can be written as the relation predicate $r(R)(x_1, \ldots, x_n)$ with $\{x_1, \ldots, x_n\} = X$. In a similar manner, join predicates j_n are comparable to the equijoin operator of the relational algebra. That means that a equijoin $r(R_1) \bowtie_{a=b} r(R_2)$ with $R_1, R_2 \subseteq R, a \in R_1$ and $b \in R_2$ can be written as the join predicate $r(R_1).a = r(R_2).b$. Finally, a selection predicate p_l represents the selection operator of the relational algebra. A selection $\sigma_F(r(R))$ with the selection condition F is written as the selection predicate

[1] The context predicates that where introduced in [6] are not considered in this paper but will be included in our approach in the future.

$r(R).F$. The selection condition F is restricted to comparisons with constants of the form `attribute` γ `constant` with $\gamma \in \{\leq, <, =, \neq, \geq, >\}$.

Thus, database queries are given in a standardized calculus notation, i.e. in conjunctive form. Predicates are ordered in a lexicographic manner: at first relation predicates r_i, then the join predicates j_k and the selection predicates p_l.

Definition 1. *A database query* $Q = \{r_1 \wedge \ldots \wedge r_m \wedge j_1 \wedge \ldots \wedge j_n \wedge p_1 \wedge \ldots \wedge p_o\}$ *can be represented as a sequence of predicates* $\langle r_1, \ldots, r_m, j_1, \ldots, j_n, p_1, \ldots, p_o \rangle$*, where* $\forall i, k \in 1 \ldots m, i < k \Rightarrow r_i \triangleleft r_k$ *and* $\forall i, k \in 1 \ldots n, i < k \Rightarrow j_i \triangleleft j_k$ *as well as* $\forall i, k \in 1 \ldots o, i < k \Rightarrow p_i \triangleleft p_k$ *holds. Here,* $\triangleleft$ *means "lexicographically smaller".*

Obviously, this query language is *not* strong relational complete, but is restricted to a subset of calculi which is sufficient for the realization of typical applications of mobile information systems.

Now, the trie can be described as follows: Each query predicate is represented as an edge and leaves represent links to mobile device ID-lists. Thus a database query $Q_P = \{r_1 \wedge r_2 \wedge \ldots \wedge r_m \wedge j_1 \wedge j_2 \wedge \ldots \wedge j_n \wedge p_1 \wedge p_2 \wedge \ldots \wedge p_o\}$ is included in the trie in form of the complete path $P = r_1 r_2 \cdots r_m j_1 j_2 \cdots j_n p_1 p_2 \cdots p_o$ from the root of the trie. A mobile device ID-list contains all IDs of mobile devices having registered the query represented by the corresponding path.

3.1 Physical Transformation of Database Queries into Trie-Paths

Due to optimization issues regarding the implementation of this index approach, we have to refine the theoretical description given above. Relation predicates r_i consist of the name of considered relation $r(R_i)$ and a set of projected attributes $(x_1, \ldots, x_n)$. As the relation name can be used by various queries that project different sets of attributes, we store the relation name and the attribute set separated as $k_{r_i} \hat{=} r(R)$ and $k_{p_i} \hat{=} (x_1, \ldots, x_n)$, respectively.

Join predicates j_i are stored undivided as $k_{j_i} \hat{=} r(R_1).a = r(R_2).b$, but we have to add relation nodes for relations, that are used in the join predicates but not in the projection predicates.

Selection predicates p_i consist of an attribute name `attribute`, a comparison operator $\gamma \in \{\leq, <, =, \neq, \geq, >\}$ and a comparative value `constant`. Obviously, an attribute name can be used in different selection predicates with various comparison operators and various comparative values. So, we represent selection predicates as two separated parts $k_{a_i} \hat{=}$ `attribute`γ and $k_{v_i} \hat{=}$ `constant`.

Furthermore, implementing a trie requires to encapsulate the information logically represented by the edges into the nodes[2]. So we have six different kinds of nodes: the root of the trie, *relation nodes* k_r, *projection nodes* k_p, *join nodes* k_j, *attribute nodes* k_a and *attribute value nodes* k_v. Figure 2 illustrates the physical implementation of the example shown in Figure 1. Furthermore, the last join node contains a list of all attributes that are used in selection predicates. This is necessary for minimizing the space consumption while checking their relevance (see Section 3.2). In order to minimize the number of nodes that are checked per update, the node order is based on the following restrictions:

[2] Nodes are implemented in form of Java classes.

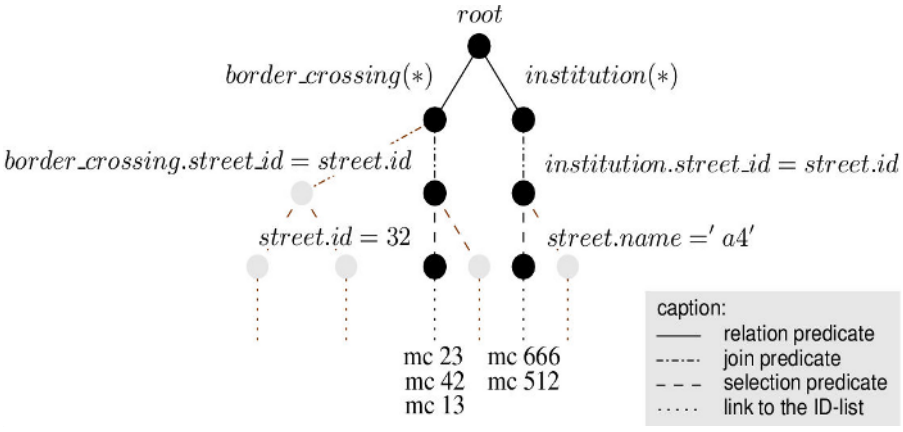


Fig. 1. Logical trie representation of queries

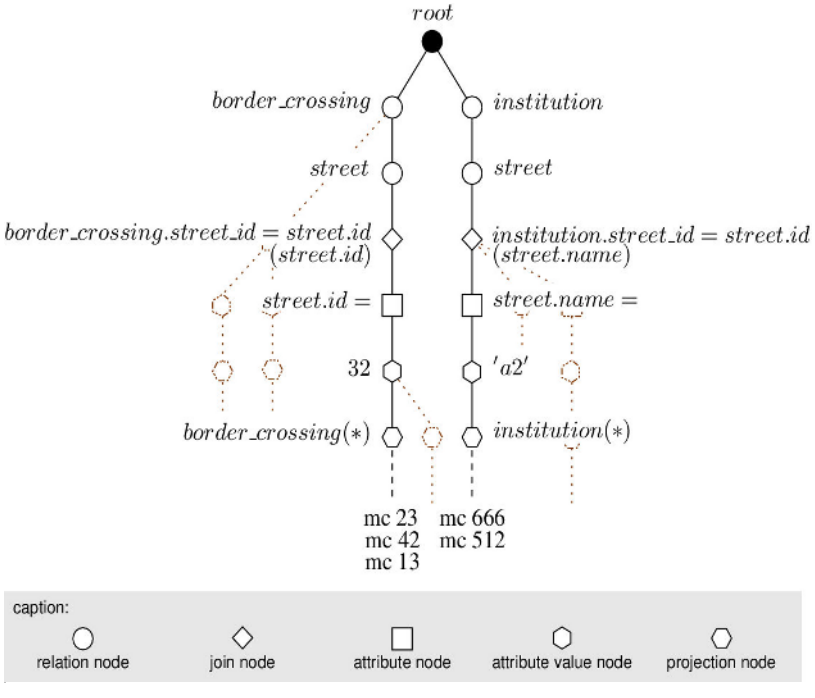


Fig. 2. Physical trie representation of queries

– Most of all relation nodes restrict the search space for an incoming update. If an update is based on relation R_1 we do not have to check queries that do not use R_1. So, relation nodes are stored directly below the root.

– As mentioned above, join predicates are used similar to the equijoin operator. So, if a registered query uses join predicates it is inevitable to compute the join. Surely, this operation is quite expensive but without executing the join, a correct check of selection predicates is impossible. So, the join nodes follow the relation nodes in a path.
– The position of the projection nodes at the bottom is motivated by the fact, that the relevance of a projection can be reduced to update operations. If we first check join predicates and selection predicates we can fully check the relevance of insert and delete operations, at this. Now, projection predicates are only relevant if the projected attribute is neither used as join attribute nor is included in the selection predicates. But, to check this we first need to look at the join nodes as well as at the attribute and attribute value nodes.

Definition 2. *The node order in a trie path is given as* $k_{r_1}, \ldots, k_{r_m}, k_{j_1}, \ldots, k_{j_n}, k_{a_1}, k_{v_1}, \ldots, k_{a_o}, k_{v_o}, k_{p_1}, \ldots, k_{p_q}$, *where* $\forall i, l \in 1 \ldots m, i < l \Rightarrow k_{r_i} \lhd k_{r_l}$, $\forall i, l \in 1 \ldots n, i < l \Rightarrow k_{j_i} \lhd k_{j_l}$, $\forall i, l \in 1 \ldots o, i < l \Rightarrow k_{a_i} \lhd k_{a_l}$ *and* $\forall i, l \in 1 \ldots q, i < l \Rightarrow k_{p_i} \lhd k_{p_l}$ *holds. At this,* $\lhd$ *means lexicographically smaller.*

3.2 Looking up the Trie

A trie look-up is performed for each incoming update *before* this update is performed on the database. The aim is to compute a list of client IDs of the mobile clients that had registered a query which is affected by this update. We distinguish between three different kinds of updates: (1) insert, (2) update and (3) delete operations. All these can only be relevant for registered queries that use the same relation as the update. So, we first compare the relation nodes with the given relation name (see Algorithm 1). At this, we benefit from the lexicographical order of the relation nodes (see line 10). The result is a set of pointers to the found relation nodes. These pointers are used as starting points for the following steps. Because of the physical structure of the trie, this step also checks the "hidden" relations that are used for join-predicates only.

Algorithm 1: *Checking relation predicates*

```
01 OUTPUT:   RN    // a set of pointers to the found relation nodes in the path
02 INPUT:    r     // name of the relation affected by this update
03           trie with root node
04
05 RN = {}
06 checking_relation_predicates(node, r, RN)
07 for each child c_i of node do
08     if c_i.value = r and c_i.type = k_r then RN = RN ∪ {c_i}; return
09     else
10         if c_i.type ≠ k_r or r ◁ c_i.value then return
11         else call checking_relation_predicates(c_i, r, RN)
```

Relation predicates can be followed in a path by a join node, by an attribute node or by a projection node. After calling Algorithm 1 we have to find out the kind of the next

node. Because of the space limitations we skip a detailed description of this here but assume that the algorithm returns the following three sets: (1) FJ is a set of the **f**irst **j**oin nodes in paths that are represented in RN, (2) FA is a set of the **f**irst **a**ttribute nodes in paths that are represented in RN but not in FJ and (3) FP is a set of the **f**irst **p**redicate nodes in paths that are represented in RN but not in FJ and not in FA.

Unfortunately, checking the join predicates in a uniform way, similar to the relation predicates, is not efficiently possible because updates, deletes and inserts modify join results in various ways. Furthermore, we have to compute the joins on the database because the relevance of the selection predicates, that is checked later on, depends on the join results. But, we do not have to consider all attributes. In fact, the temporary join result contains only the join attributes and the attributes used in the selection predicates.

Checking Join Nodes. Checking the relevance of an update operation regarding join nodes is done for each element of the set FJ. The return value of each call is a set AJ_i of pointers to the nodes below the join nodes and PR_i a set of temporary join results, that are needed for checking selection predicates. All paths that are not represented in the union of all AJ_i are not considered in the following steps. As mentioned above join nodes contain join predicates of the form $r(R_1) \bowtie_{a=b} r(R_2)$ with $R_1, R_2 \subseteq R$, $a \in R_1$ and $b \in R_2$. The algorithm first collects all join nodes of a path. So, we get a join-statement for each path of the form $r(R_1) \bowtie_{a_1=a_2} r(R_2) \bowtie_{a_3=a_4} r(R_3) \cdots r(R_j) \bowtie_{a_{2j-1}=a_{2j}} r(R_{j+1})$ with $j \in \mathbb{N}$ and $a_{2j-1} \in R_j$. Checking the relevance of such a statement for the update is done in the following way:

Insert Operation: We assume inserts in standard SQL-notation: `INSERT INTO table_name (column_list) VALUES (value[,...])`.

$A = (a_1^i, \ldots, a_n^i)$ is the attribute list of relation used for inserting data. $V = (x_1^i, \ldots, x_n^i)$ is the tuple of inserted values. Furthermore, $r(R_{j+1})$ is the relation that is used for inserting the data, $a_1^i = a_{2j}$ is the according join attribute and x_1^i is the inserted value of a_1^i. So the join predicates are affected by the update if $\pi_{a_1}(r(R_1)) \bowtie_{a_1=a_2} \pi_{a_2,a_3}(r(R_2)) \ldots (\sigma_{a_{2j-1}=x_1^i}(r(R_j))$ is not empty.

Delete Operation: Delete operation affect a join predicate if the tuples that have to be deleted are included in the join result. We assume delete operations in SQL-notation[3] as `DELETE FROM table_name WHERE clause`. Because this can affect more than one tuple in the database we first have to look up the according values of the join predicate. With the updated relation $r(R_{j+1})$, we can use the `clause` that was given by the statement: $r(R_T) = \pi_{2j}(\sigma_{\texttt{clause}}(r(R_{j+1}))$. Now, the join is affected by the update if $\pi_{a_1}(r(R_1)) \bowtie_{a_1=a_2} \pi_{a_2,a_3}(r(R_2)) \cdots \pi_{a_{2j-2}}, \pi_{a_{2j-1}}(r(R_j)) \cap r(R_T)$ is not empty.

Update Operation: Currently we handle update operations as combination of delete and insert operations.

As aforementioned, we need the result of the joins to check the selection predicates. Therefore, attributes that are not used as join attributes may be required. So, we have to guarantee that these attributes are included into the temporary result. In fact, we do not

[3] Currently we forbid the usage of cascading delete operations.

use the minimal join presented above but add all attributes of selection predicates, that are included in the last join node, to the projections.

Checking Attribute Nodes and Attribute Value Nodes. First we look up the selection predicates with a recursive algorithm. It returns for each relevant path a set of selection predicates P_Q. An insert operation affects a query Q if it satisfies at least one selection predicate. Formally, that means for an inserted tuple $A = (a_1^i, \ldots, a_n^i)$ with the values $V = (x_1^i, \ldots, x_n^i)$ that $\exists p \in P_Q | \sigma_p(r(R_A)) \neq \emptyset$ must hold. In order to check the relevance of delete operations we have to distinguish between queries, that use join nodes and queries without join nodes. In the first case, we have to check whether the delete affects the part of the according temporary join result that is covered by the selection predicates. Therefore, with an SQL-notated delete, the temporary join result $TJ \in PR_i$ and the disjunction $D = p_1 \vee p_2 \vee \ldots \vee p_o$ with $o = |P_Q|$ that means, that $\sigma_{\texttt{clause}}(\sigma_D(TJ)) \neq \emptyset$ must hold. In the second case we have to use the base relation instead of TJ. Therewith, the selection predicates of queries without join predicates are affected by a deletion operation if $\sigma_{\texttt{clause}}(\sigma_D(r(R_{\texttt{table\_name}}))) \neq \emptyset$ holds. Updates are handled as combination of delete and select operation, again.

Checking Projection Nodes. We do not have to check projection nodes or projection attributes, respectively, if the update operation is an insert or an delete because these operation increase or decrease the cardinality of the query result. So, the relevance of such updates is already recognized by checking the selection predicates and/or join predicates. But, in the case of updates it can occure, that the update modifies join attribute values and a selection attribute values but not the projected attributes. Such updates are not relevant for a registered query if the projected attributes are not contained in the list of updated attributes.

Fetching the IDs of the Mobile Clients. If all checks result in a relevance of an update operation we fetch the IDs of the mobile clients and notify them about the update. However, we do not consider the update of the data that is managed on the mobile clients but will do this in future work.

4 Evaluation

To evaluate our approach we implemented a small driver support systems that provides traffic information about road works, traffic jams as well as additional information about public utilities in a location depended manner. Here, we do not consider updating the trie by fast moving cars but approximate journeys by locating cars on a street. That means, that streets are implemented as a line between two coordinates. In fact, the benefit of our approach is not the complete realization of such a system but we use it for evaluating the update notification. Typical queries are: “Where is the next parking block with available parking lots?”, “Is there a road block on my current road?” or “Where is the next garage?”. The corresponding database[4] is realized using PostgreSQL and contains three cities and

[4] See [7] for details.

about 200 fictitious streets. Some streets cross cities. Furthermore we inserted about 9000 public utilities distributed among the cities. We assume 20 permanent traffic jams and five border crossings that hamper the traffic.

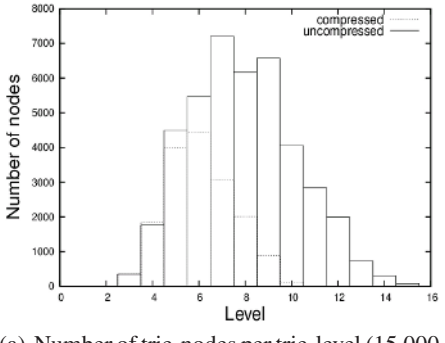
Queries are generated automatically and contain 1-3 projection predicates, 0-2 join predicates and 0-3 selection predicates. Two examples for such queries are:

- $\langle$*border_crossing(*), border_crossing.street_id* $=$ *street.id*, *street.id* $= 32\rangle$
- $\langle$*institution(id)*, *institution.street_id* $=$ *street.id*, *street.name* $=' A4'\rangle$

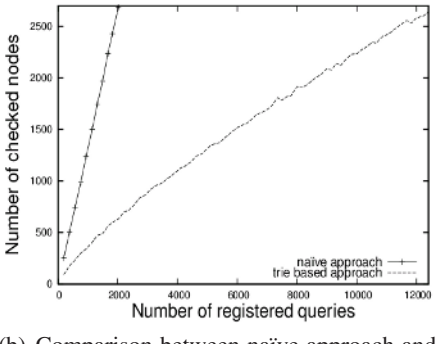
As mentioned in Section 3.2 we use updates in standard SQL notation, like:

- `DELETE FROM R_WORKS__T_JAMS WHERE street_id=22 AND gps_start_y=1700`
- `UPDATE BORDER_CROSSING SET w_t_freight_vehicles='03:54' WHERE name='Mittenwald'`

Figure 3(a) illustrates height and width of an example trie that represents 15,000 queries whereby 12,766 queries are different from each other. At this, we also included the values for a compressed representation that utilizes the fact, that values of nodes with only one child node can be stored in one node. While processing our algorithms that only means, that checks regarding this two values use the same node pointer but the space consumption is much lower.



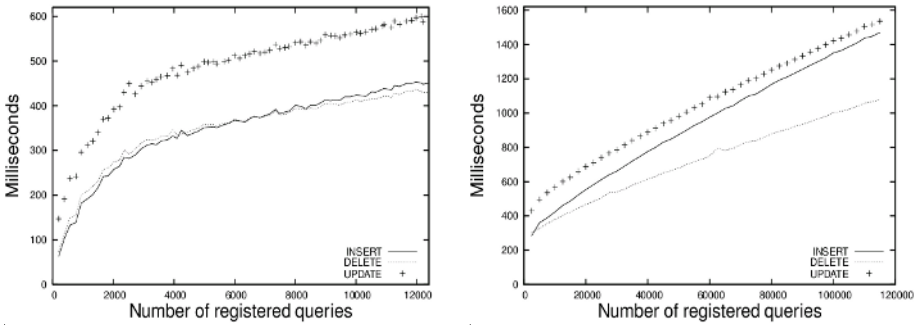
(a) Number of trie-nodes per trie-level (15,000 queries; 12,766 different queries)



(b) Comparison between naïve approach and the trie based approach

Fig. 3. Space consumption and performance evaluation

At first we compare our approach to the naïve approach that represents the queries in a list. That means, the naïve algorithm sequentially scans the registered queries. The result of this comparison is shown in Figure 3(b). The predictable large number of nodes checked in the naïve approach depends on the fact, that such approaches typically do not consider predicate overlapping between the queries of different users. The result of our approach for this test depends on the trie representation used for storing the queries. So, we can point out that - in this test - our approach performs better than the naïve approach.



(a) up to 12,000 registered queries (b) up to 120,000 registered queries

Fig. 4. Correlation between duration of updates and number of registered queries

The trie and its algorithms are implemented under Windows XP in Java. The communication between Java and PostgreSQL is realized with the standard PostgreSQL-JDBC-driver. For the experiments we used a standard PC with an AMD Athlon™XP 2100+ processor and 768 MB Ram. The duration of handling an update on this configuration is illustrated in Figure 4. At this, Figure 4(a) shows, that insert and delete operations are not as expensive as update operations. But in Figure 4(b) we see that the curves of insert and update operations converge with an increasing number of registered queries.

5 Conclusions and Outlook

In this paper we presented an indexing scheme for update notification in large mobile information systems. Queries posed by mobile clients are represented as paths in a trie at the server. We first discussed the used query representation that uses queries in form of conjunctively connected relation, selection and join predicates. Afterwards, the implementation and the physical representation of the trie was introduced. We illustrated how nodes are fetched and how the relevance of an update is checked regarding the different predicates. Finally, we presented and evaluated our approach and pointed out its benefit.

In spite of the acquired good results, there is a lot of future work. First of all, the used query language is not relational complete. We currently do not support unions. In addition to this, aggregation functions are not supported because of the used calculus. We also skipped the context predicates that are mentioned in [6]. In fact, first steps to support large context based mobile information are done. In [5] we introduced a general model that is not limited to location based queries but allows to specify more context elements like time relevance, task dependency, et cetera. Moreover, we plan to optimize the query index. For example, selection predicates are currently represented in a redundant manner, so we hopefully benefit from storing them in a clustered way or as intervals similar to 1-dimensional R-Trees.

References

1. Peter W. Foltz and Susan T. Dumais. Personalized Information Delivery: An Analysis of Information Filtering Methods. *ACM CACM*, 35(12):51–60, December 1992.
2. E. Fredkin. *Trie memory*. Information Memorandum, Bolt Beranek and NewMan Inc., Cambridge, MA, 1959.
3. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. *SIGMOD Record*, 25(2):173–182, 1996.
4. Sha Guo, Wei Sun, and Mark Allen Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM TODS*, 21(2):270–293, 1996.
5. H. Höpfner and K.-U. Sattler. Semantic Replication in Mobile Federated Information Systems. In A. James, S. Conrad, and W. Hasselbring, editors, *Proc. of the Fifth Workshop EFIS*, pages 36–41. Akademische Verlagsgesellschaft Aka GmbH, Berlin, July 2003.
6. Hagen Höpfner and Kai-Uwe Sattler. SMoS: A Scalable Mobility Server. In Anne James and Muhammad Younas, editors, *Poster Proc. of BNCOD20* , pages 49–52. Coventry University, jul 2003.
7. Hagen Höpfner, Stephan Schosser, and Kai-Uwe Sattler. Toward Trie-based Indexing of Mobile Clients in Large Mobile Information Systems. Preprint 02, Otto-von-Guericke-University of Magdeburg, Department of Computer Science, January 2004. http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/preprints/2004/HoeSchSat.html.
8. Ken C. K. Lee, Hong Va Leong, and Antonio Si. Incremental View Maintenance for Mobile Databases. *Knowledge and Information Systems*, 2(4):413 – 437, nov 2000.
9. S. H. Phatak and B. R. Badrinath. Multiversion Reconciliation for Mobile Databases. In *Proc. of the 15th ICDE, Sydney, Austrialia*, pages 582–589. IEEE Computer Society, March 1999.
10. Shirish H. Phatak and B.R. Badrinath. Transaction-centric Reconciliation in Disconnected Databases. *ACM Monet Journal*, 53, 1999.
11. Evaggelia Pitoura and George Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, Dordrecht, 1998.
12. X-H. Sun, N. N. Kamel, and L. M. Ni. Processing Implications on Queries. *IEEE Transactions on Software Engineering (SE)*, 15(10), October 1989.
13. C. Türker. *Semantic Integrity Constraints in Federated Database Schemata*. Akademische Verlagsgesellschaft Aka GmbH, Berlin, 1999.
14. Ugur Cetintemel, Michael J. Franklin, and C. Lee Giles:. Self-Adaptive User Profiles for Large-Scale Data Delivery. In *Proc. of the 16th ICDE*, pages 622–633. IEEE Computer Society, March 2000.