

Towards Trie-Based Query Caching in Mobile DBS*

Hagen Höpfner and Kai-Uwe Sattler

Otto-von-Guericke-University of Magdeburg
Department of Computer Science
Institute of Technical and Business Information Systems
P.O. Box 4120, 39016 Magdeburg, Germany
{hoepfner | kus}@iti.cs.uni-magdeburg.de

Abstract: The usage of mobile equipment like PDAs, mobile phones, Tablet PCs or laptops is already common in our current information society. Typically, mobile information systems work in a context dependent (e.g. location dependent) manner which means that queries differ mostly only in some context-related predicates on the same set of relations. Considering this characteristics, query processing on mobile devices can take benefit from results of previously performed queries by using a query cache. In this paper, we describe such a caching approach based on a trie structure and organized by the query predicates which are associated with the corresponding result sets. We present the cache structure and the process of query rewriting as well as discuss implementation issues.

1 Introduction and Motivation

On stationary computers managing large and complex data sets is often done by database management systems. Due to the increasing complexity of mobile information systems, DBMS-techniques on mobile devices become more important. Though mobile devices are quite powerful, compared to stationary computers they are called “lightweight”. Nearly all available enterprise databases (e.g. Oracle, DB2) are offered as functionally limited lite versions usable on lightweight devices [IBM02, Ora02b, iAn03, IBM03].

Today’s mobile radio technologies are still slow and expensive. Therefore, time based payment models are counterproductive considering a permanent connection. But also volume based payment models associated with large data sets cause high costs. Furthermore, technical problems (net overload, electrostatic shield, etc.) can result in disconnections. Thus, there is the need for storing necessary data directly on the mobile hosts.

In mobile information systems such data sets (replica sets) are requested by the user via the mobile device from a database server [Gol03] and afterwards locally stored at the mobile device. Therefore, locally available replica sets can be considered as *cache*. New queries

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

should reuse the cached data in order to minimize wireless (expensive and slow) communication. Only locally unavailable data should be requested from the database server. Designing such a cache comprises two central tasks:

Query Rewriting: Queries, which are executed on the mobile system, have to be rewritten in order to use the cache. If a query is not completely answerable by the cache, it has to be decomposed into sub-queries in such a manner that only a minimal subset of the complete result must be requested from the database server.

Cache Replacement: If there is no more available memory on the mobile device, no more replica sets can be cached. Therefore, parts of the cache contents have to be deleted. The decision to remove a subset of the cached data must consider possible future queries. That means, the remaining state of the cache must fit upcoming queries as good as possible.

In this paper we present first ideas of a cache which uses a trie¹ indexing schema to provide an efficient access to the cached replica sets. We take advantage of the special characteristics of mobile database systems:

- In mobile environments queries are mostly executed in a context dependent² manner. That means that queries in mobile information systems reflect the elements of the context (e.g. location, time). For example, a person, who is visiting London and is using a mobile information system offering information about cultural events, will, in all likelihood, query current (time dependency) information of events performed in London (location dependency). Such systems are often called “location dependent services”. But the context of a user contains a lot of other elements [LH01] (e.g. used hardware, group membership, task to handle).
- Mobile information systems consider the high cost of wireless data transmission. Therefore, a lot of “small” queries are used, which can be answered by a “small” but exact result set. If the user needs more detailed information, additional refined queries are issued.

We use these special characteristics of queries in mobile information systems for indexing the cache on a mobile device but consider only queries which must not include aggregate functions or joins. In addition, we do not consider here cache replacement and cache coherency. Cache coherency is a big problem especially in mobile environments but we postpone this aspect to future work. In the face of replacing cache entries with new data our approach is compatible with known techniques like LFU (least frequently used) or LRU (least recent used). Therefore, the cache entries have to be enhanced by reference counters or timestamps, respectively. But, having regard to an extended query language as presented in [HS03b] we can denote support for context dependent techniques here.

¹A trie [Fre59] is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes (see also [Bla98]). In our case, we replace string prefixes by query prefixes (conjunctions of predicates).

²based on the context of the user and/or the context of the mobile device

There are several publications (e.g. [DFJ⁺96], [GG99]) demonstrating that semantic caches are more efficient than time or reference based caching approaches. In mobile database systems the most important approach of semantic caching is the LDD-cache [RD00] (based on [DK98]). In [RD99] a cluster extension of the LDD-cache is presented which increases its performance. In contrast to this approach we provide an index structure that allows to access cached information more efficient.

The remainder of this paper is structured as follows: After presenting the index and cache structure in Section 2, we discuss our query processing and query rewriting approach in Section 3. We also compare our trie-based query-cache to the LDD-cache in Section 3. Finally, we discuss different implementation variants in Section 4. Because of the given space limitations we do not discuss any cache replacement strategy in this paper.

2 Query Cache Structure

The main issue of a query cache is the granularity of the cache entries. In contrast to normal database management system caches, which store logical pages, query caches store query results (relation or fragments of relations respectively). Therefore, the queries are used as access keys. Now it is possible to check new queries, whether they can be answered from the cache. At this, we distinguish between the following four cases:

1. The same query has been performed already and its result is still stored in the cache.
2. A query, which results in a superset of the requested result, has been performed already and its result is still stored in the cache. Therefore, the new query can be locally answered by using additional operations (e.g. selection). Case (1) can be considered as a special case of this case.
3. A cached query and a new query overlap, i.e., a subset of the result can be answered by the cache. Only the non-overlapping subset must be requested from the database server.
4. There is no compatible query in the cache.

Furthermore, we have to take into account, that a new query may be answered by a combination of cache entries. Regarding to these requirements our cache structure has to support an efficient search for appropriate entries.

Based on this and on the special characteristics of mobile information systems we chose the following cache structure: A cache entry always corresponds to a fragment of a relation – thus to a set of tuples. The entries are indexed by a trie structure which represents the query. For that purpose we represent queries in a standardized calculus notation, i.e. in conjunctive normal form. Predicates are ordered in a lexicographic manner: at first relation predicates r_i , then the selection predicates p_j (also lexicographically ordered). Relation predicates describe the considered schema of a relation. For example, `buildings(name)` corresponds to `SELECT name FROM buildings`. Selection

predicates corresponds to a WHERE-clause, respectively (e.g. `buildings.x>200` is a predicate representation of `WHERE buildings.x>200`).

A query $Q = \{r_1 \wedge r_2 \wedge \dots \wedge r_m \wedge p_1 \wedge p_2 \wedge \dots \wedge p_n\}$ can be represented as a sequence of predicates $\langle r_1, r_2, \dots, r_m, p_1, p_2, \dots, p_n \rangle$, where $\forall i, j \in 1 \dots n, i < j \Rightarrow p_i \triangleleft p_j$ as well as $\forall i, j \in 1 \dots m, i < j \Rightarrow r_i \triangleleft r_j$ (\triangleleft means “lexicographically smaller”) holds. Obviously, this query language is *not* strong relational complete, but is restricted to a subset of calculi which is sufficient for the realization of context based, mobile information systems.

For a trie that means, that edges represent predicates and nodes represent links to the caches fragments. A path $P = r_1 r_2 \dots r_m p_1 p_2 \dots p_n$ in the trie (from the root to any node) corresponds to the query $Q_P = \{r_1 \wedge r_2 \wedge \dots \wedge r_m \wedge p_1 \wedge p_2 \wedge \dots \wedge p_n\}$. Thus, the result of a query Q_P represented by the path P can be found in the cache as the entry addressed by P .

This cache approach is not limited to exact query matches. We support also the other two cases (see above) by following a path in the trie predicate by predicate. If a node linking a fragment is reached and if there are more predicates in the new query, the new query can be answered by running it on the linked fragment. Otherwise, if no node linking a fragment is reached so far but all predicates of the new query were used, this new query can be partially answered by combining all fragments linked by child nodes of the current one. Therefore, the query processor must generate a *compensation query* which request the missing sub result and combines it with the cached sub result.

3 Cache-supported Query Processing

After the presentation of cache and index structures in the previous section we discuss the cache-supported query processing in the following. The central point is the query rewriting phase: the problem of splitting a query into two sub-queries in such a manner, that a local sub query can be performed on the cache and a remote query (*compensation query*) can be sent to the database server and performed there. The main focus is the minimization of the amount of data retrieved from the database server and thus the minimization of transfer costs.

Query rewriting is based on correspondence relations between queries. Following the four cases presented in the previous section we can distinguish correspondence relations of a query P and another query Q as follows. For simplification we assume for now, that P and Q contain the same relation predicates, i.e. only the selection predicates are taken into account.

(A) $P \equiv Q$ means, that the query can³ be directly performed on the cache. Therefore, obviously equivalence of all predicates must hold:

$$\langle p_1, \dots, p_n \rangle \equiv \langle q_1, \dots, q_n \rangle \Leftrightarrow \forall i = 1 \dots n : p_i \equiv q_i$$

³granted that all attributes used by the filter condition are included into the cached result

(B) $P \sqsubset Q$ means, that Q can be answered by the cache, but an additional filter condition $filter(P, Q)$ is required. This case can be distinguished into the following sub cases:

- (a) Q consist of more predicates: $\langle p_1, \dots, p_m \rangle \sqsubset \langle q_1, \dots, q_n \rangle \Leftrightarrow m \leq n \wedge \forall i = 1 \dots m : p_i \equiv q_i$. The corresponding filter condition is:

$$filter(P, Q) = \langle q_{m+1}, \dots, q_n \rangle.$$

- (b) Q consist of *more restrictive* predicates than P . A predicate q is more restrictive than a predicate p (both defined on an attribute A) if q holds for less tuples than p whereas a domain wide equipartition of the attribute values is assumed. For instance, $a > 50$ is more restrictive than $a > 10$, written as $a > 50 \prec a > 10$. Operators used for comparison strongly depend on the involved attribute types (coordinates, time, date, integer, etc.). In [HS03a] we have presented an approach, that uses type dependent “distance” functions (so called ξ -functions). These functions can also be used in order to compare the restrictivity of selection predicates.

$$\langle p_1, \dots, p_m \rangle \sqsubset \langle q_1, \dots, q_n \rangle \Leftrightarrow \forall i = 1 \dots k < n : \\ p_i = q_i \wedge \forall j = k + 1 \dots n : q_j \prec p_j$$

Thus, the filter condition is the combination of the more restrictive predicates:

$$filter(P, Q) = \langle q_{k+1}, \dots, q_n \rangle$$

(C) $P \sqsupset Q$ means, that Q consists of less predicates or Q is less restrictive than P . As a result Q can be partially answered by the cache. For getting the whole result, a compensation $compensation(P, Q)$ must be performed on the database server. This case can be distinguished into two sub-cases, too:

- (1) $\langle p_1, \dots, p_n \rangle \sqsupset \langle q_1, \dots, q_m \rangle \Leftrightarrow m < n \wedge \forall i = 1 \dots m : p_i = q_i$ Therefore, the result of a compensation query has to contain all tuples that are not included in the result of P but belong to the result of Q :

$$compensation(P, Q) = \langle q_1, \dots, q_m, \neg p_{m+1}, \dots, \neg p_n \rangle$$

- (2) $\langle p_1, \dots, p_n \rangle \sqsupset \langle q_1, \dots, q_n \rangle \Leftrightarrow \forall i = 1 \dots k < n : p_i = q_i \wedge \forall j = k + 1 \dots n : p_j \prec q_j$ Thus, the compensation query is:

$$compensation(P, Q) = \langle q_1, \dots, q_n, \neg p_{k+1}, \dots, \neg p_n \rangle$$

(D) $P \not\equiv Q$ means, that there are no consensuses between the predicates of the queries.

$$\forall i = 1 \dots k < n : p_i \not\equiv q_i$$

Thus, the cache entries belonging to P are not usable for answering Q .

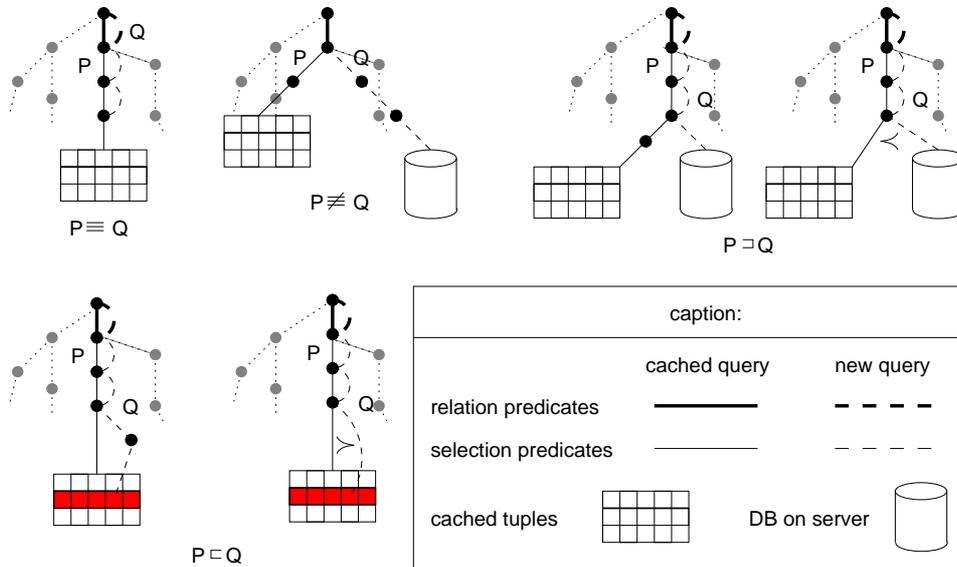


Figure 1: Relations between a cached query Q and a new query P

Considering this correspondence relations (see also Figure 1) we are able to modify a given query in order to use the cache now. The idea is to collect the nodes, that link appropriate cache entries, while passing the trie. The result of this procedure are candidate sets of nodes corresponding to the cases mentioned above. Because we can consider case (A) as case (B) with an empty filter condition, Algorithm 1 uses only two sets CS_{AB} and CS_C . If both sets are empty, case (D) is given. The trie is traversed by using the recursive procedure *find_candidate_nodes* shown as algorithm 2. The generation of the candidate sets is followed by handling the cases. The function *path* used in the algorithm returns the query which corresponds to the path from root of the trie to a given node.

Algorithm 1: *trie-based Rewriting*

given:

query Q
 trie with *root* node

$CS_{AB} := \{\}$
 $CS_C := \{\}$

find_candidate_nodes (0, *root*)

if $CS_{AB} = \{\} \wedge CS_C = \{\}$ **then**

perform Q remotely

else if $CS_{AB} \neq \{\}$ **then**

```

Node  $n \in CS_{AB}$ 
 $P := path(n)$ 
 $Q' := filter(P, Q)$ 
perform  $Q'$  on the cache-entry associated with  $n$ 
else if  $CS_C \neq \{\}$  then
   $n := min\_cost(CS_C)$ 
   $P := path(n)$ 
   $Q' := compensation(P, Q)$ 
  do  $Q' \cup$  cache-entry associated with  $n$ 
fi

```

Because there may exist more than one candidate node which could be used for answering a query, the best node have to be detected (see below). Afterwards the filter and the compensation query is computed and performed.

Algorithm 2 implements the traversing of the trie. Starting at the root node all edges, i.e. all predicates, will be checked. If a predicate p in the trie is equal to or less restrictive than the corresponding predicate of the new query, all child nodes $child_p(n)$ are tested. But if both predicates are contradictorily, the search will stop. If none of these alternatives hold⁴ then the predicate is ignored and the search continues with all child nodes.

If a checked node contains a link to a cache entry, the path and thus the query is reconstructed first. Afterwards the correspondence relation between the query and the given new query is computed incrementally. This is possible because the predicate has been checked already while running the algorithm. For the simplification of the presentation we skip a more detailed discussion of this aspect here. If a consensus is found, the actual node is added to one of the two candidate sets CS_{AB} or CS_C , respectively. If $P \equiv Q$ holds, the search stops immediately.

Algorithm 2: Traversing the trie

```

find_candidate_nodes (level, node)
if cache-entry associated with node available then
   $P := path(node)$ 
  if  $P \equiv Q$  then
     $CS_{AB} := \{ node \}$ 
    return
  else if  $P \sqsubset Q$  then
     $CS_{AB} := \{ node \}$ 
  else if  $P \sqsupset Q$  then
     $CS_C := CS_C \cup \{ node \}$ 
fi

```

⁴this case corresponds to $P \sqsupset Q$

```

forall  $p \in \text{predicates}(n)$  do
  if  $p \equiv q_{\text{level}} \vee q_{\text{level}} \succ p$  then
     $\text{find\_candidate\_nodes}(\text{level}+1, \text{child}_p(\text{node}))$ 
  else if  $\neg(p \wedge q_{\text{level}})$  then
    /* unaccomplished condition */
    return
  else
    /* ignore predicate p */
     $\text{find\_candidate\_nodes}(\text{level}+1, \text{child}_p(\text{node}))$ 
  fi
od

```

A path is chosen cost dependently from the set of alternative paths. A complete estimation of all costs (e.g. based on cardinalities or histograms) causes an essential additional expense. Because of the light-weighty of mobile devices only the maximal size of cached fragments is taken into account.

So far we have assumed that $P = \{r_1 \wedge r_2 \wedge \dots \wedge r_m \wedge p_1 \wedge p_2 \wedge \dots \wedge p_n\}$ and $Q = \{k_1 \wedge k_2 \wedge \dots \wedge k_u \wedge q_1 \wedge q_2 \wedge \dots \wedge q_v\}$ differ only in selection predicates, i.e. $m = u$; $\forall i \in 1 \dots m, r_i = k_i$.

Based on the previous ideas, now we discuss shortly the consequences of allowing differences between relation predicates. That means that queries can contain joins, too. We assume that R is the set of relation predicates used by P and K is the set of relation predicates used by Q . Thus, there exist four cases:

- (E) $R \cap K = \emptyset$, means, that both queries use different relations. Q must be sent to the database server.
- (F) $R = K$, means, that both queries use the same relations. Q could be answered locally by the cache but a detailed check of the selection predicates is required (see above).
- (G) $R \subset K$, means, that the new query uses Q additional relations. A compensation query must be generated in order to query the missed tuples from the database server. The join can be computed on the mobile device.
- (H) $K \subset R$, means, that the new query Q uses some of the relations used by R . Q could be answered locally by the cache but a detailed check of the selection predicates is required (see above).

For definitely extending the presented approach by allowing differences between relation predicates we have to modify algorithm 1 accordingly.

h_id	name	phone	fax	address
1	St. Maria Hospital	3920811	4433276	P.O. Box 1100
2	University Hospital	1237654	9912347	18 Avenue of The Stars
3	Dr. Koch Hospital	8877365	1253432	5 Third Avenue

Table 1: The Relation hospitals

s_id	name	form	phone	address
1	Ada Lovelace School	Grammar School	234555366	12 Kent Street
2	1st Primary School	Primary School	355355534	35 East Avenue
3	Ascaneum	Grammar School	234234234	141 11st Street
4	ECOLE	Primary School	445613334	45 Church Place
5	Gustav-Nitsche School	Secondary School	464663698	1 Nitsche Street

Table 2: The Relation schools

b_id	name	x	y
1	Ada Lovelace School	1343	2345
2	1st Primary School	442	1234
3	Ascaneum	5453	11
4	ECOLE	166	23
5	Gustav-Nitsche School	46667	98883
6	St. Maria Hospital	811	4476
7	University Hospital	7654	99147
8	Dr. Koch Hospital	8365	1342

Table 3: The relation buildings: Location of the Buildings

An Example for Illustration

We now illustrate our approach (without an explicit usage of join predicates) with a small example. We assume the following infrastructure database consisting of two relations:

1. hospitals (h_id, name, phone, fax, address)
2. schools (s_id, form, name, phone, address)

In the following we use fictive example data for an fictive example city (cf. Table 1 and Table 2). Furthermore we know that the location of each building is stored in a third relation buildings (b_id, name, x, y) (cf. Table 3).

This database is queried via a mobile client. Therefore, we assume the following sequence of – already lexicographically ordered – queries:

Q_1 : hospitals(address, name)

Q_2 : schools(address, phone) \wedge schools.name='ECOLE'

Q_3 : buildings(name) \wedge buildings.x>200 \wedge
 buildings.x<8000 \wedge buildings.y>1000 \wedge buildings.y<3000

The resulting state of the cache, which depends on the execution of these queries is illustrated in Figure 2. For the sake of clearness we did not include the values of the attributes used by the selection predicates into the result here. But these values are required for using filter queries locally.

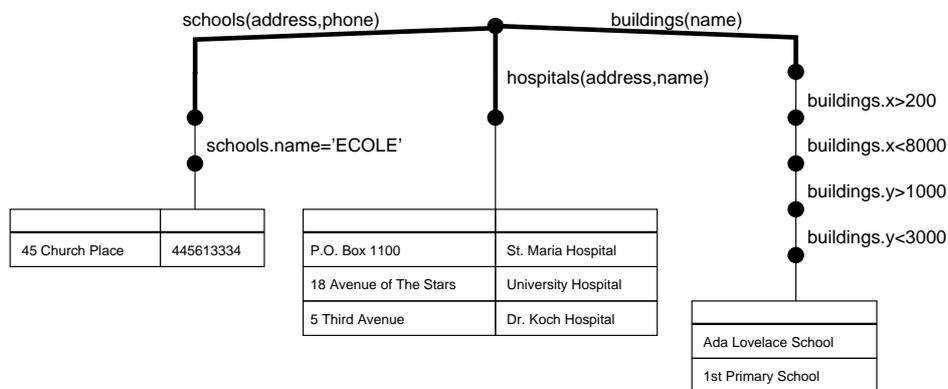


Figure 2: Example: Cache State after the Execution of Q_1 , Q_2 and Q_3

It is easy to understand how this trie was built. But now this cache can be used for answering new queries. Lets assume the following new queries:

Q_4 : hospitals(address, name) \wedge
 hospitals.address='5 Third Avenue'

Q_5 : `buildings(name) ∧ buildings.x>200 ∧ buildings.x<8000 ∧ buildings.y>1000 ∧ buildings.y<2000`

Q_6 : `schools(address, phone)`

Q_7 : `buildings(name) ∧ buildings.x>200 ∧ buildings.x<8000 ∧ buildings.y>1000 ∧ buildings.y<5000`

Obviously, we have the following relationships between the new queries and the already cached queries:

- $Q_4 \sqsubset Q_1$: Q_4 covers a subset of the result of Q_1 because it consists of more selection predicates
- $Q_5 \sqsubset Q_3$: Q_5 covers a subset of the result of Q_3 because a selection predicate is more restrictive (`buildings.y<2000` \prec `buildings.y<3000`)
- $Q_6 \supset Q_2$: Q_6 covers the results of Q_2 but may cover additional tuples because Q_6 consists of less selection predicates.
- $Q_7 \supset Q_3$: Q_7 covers the results of Q_3 but may cover additional tuples because a selection predicate is less restrictive (`buildings.y<5000` \succ `buildings.y<3000`)

So, the execution of Q_4 and Q_5 requires the usage of additional filter queries:

- $filter(Q_1, Q_4) = \text{hospitals.address} = '5 \text{ Third Avenue}'$
- $filter(Q_3, Q_5) = \text{buildings.y} < 2000$

On the other hand, the execution of Q_6 and Q_7 requires the usage of compensation queries for completing the results:

- $compensation(Q_2, Q_6) = \text{schools(address, phone)} \wedge \text{schools.name} <> 'ECOLE'$
- $compensation(Q_3, Q_7) = \text{buildings(name)} \wedge \text{buildings.x} > 200 \wedge \text{buildings.x} < 8000 \wedge \text{buildings.y} > 1000 \wedge \text{buildings.y} \geq 3000 \wedge \text{buildings.y} < 5000$

Figure 3 illustrates the effects of the usage of the filter respectively the compensation queries during the execution of Q_4 , Q_5 , Q_6 and Q_7 .

Comparative Discussion:

In the following we compare our trie-based query cache to the classical online approach without client-side caching and to a semantic caching approach for mobile environments.

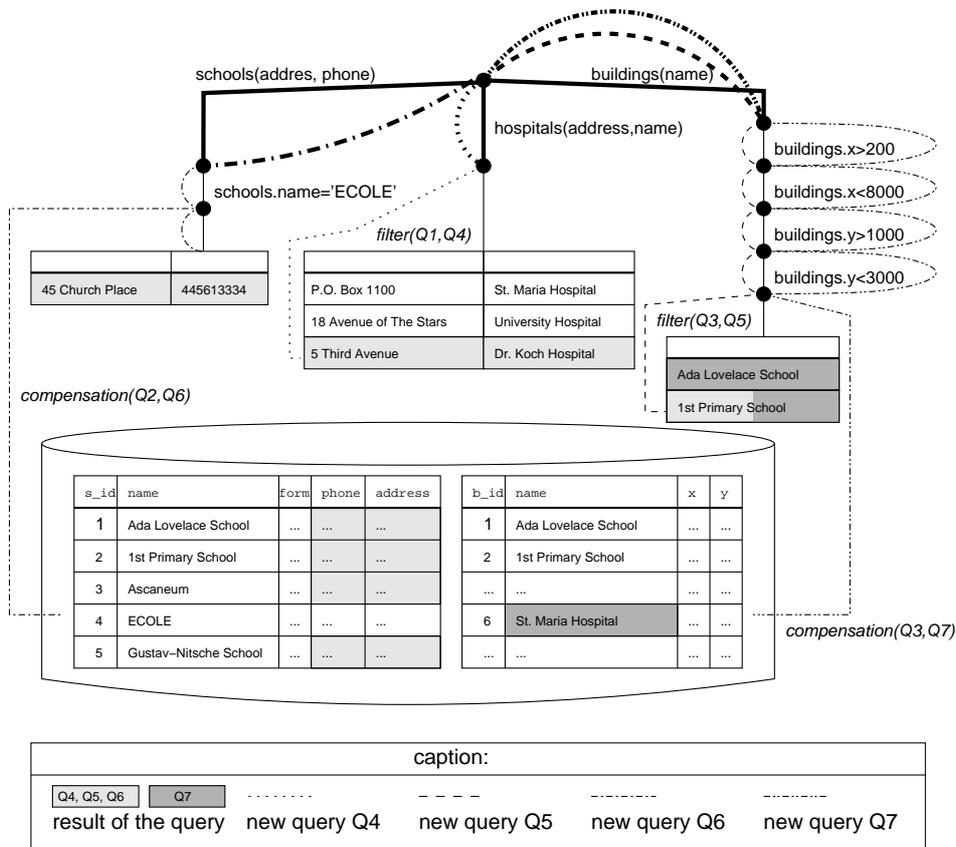


Figure 3: Example: Using Filter and Compensation Queries

Without caching all queries are submitted to the database server where they are performed. Afterwards the results are sent back to the mobile device. For the comparison to semantic caching we fall back on the LDD cache [RD00] that uses a location based and time based indexing of data which is stored as logical pages. Every index entry consist of the name of the used relation⁵, the names of the queried attributes, the appropriate location from which the query was started and a timestamp.

Algorithms for caching and retrieving data require additional computation power. Because light-weighty is the central characteristic of mobile devices, the size of the cache is strictly limited. Thus, we do not discuss an explicit derivation of all individual complexities.

A more important point is the efficiency of the cache. A cache which prevents the expensive and slow wireless data submission is more efficient than a cache which sends a lot of queries to the database server. In the online approach without a cache obviously all queries must be send to the database. Thus, it is inefficient. The LDD cache as well as our

⁵The approach can only use single-relation-queries

trie-based query cache are more efficient. The differences between these two approaches are the cache structure and the index structure and how they can be used. The LDD cache stores one separate segment per query. Our trie-based approach does it in the same way but allows, based on its special index structure, an easy merging of semantically neighboring or overlapping cache entries. Furthermore, we facilitate the splitting of join operations in order to handle relation spanning queries at least partially local. Thus, our approach is more efficient considering the need for wireless communication.

Another advantage of our approach is the flexibility considering the context elements. The LDD cache is restricted to location dependency and can handle timestamps. Because of using a trie for indexing cache entries we are able to reflect each context element directly in our index and use it for retrieving adequate fragments in the cache.

Obviously, this is only a vague and preliminary comparison but it shows that it makes sense to follow up the ideas presented in this paper. A detailed cost model for caching in mobile environments would go beyond the scope of this paper, but it is currently under work.

4 Implementation Issues

Indexes on the cache of mobile devices are used to minimize the amount of data to be transferred and the memory required. These targets can be derived directly from the specific requirements of mobile information systems, i.e. slow and expensive data transfer and limited resources of the mobile devices.

Especially, to handle the case described in Section 2, where the cache already contains a superset of the results of a new query, the client has to be able to process a query from previously cached results. Therefore, the fragments are stored using a local DBMS. On the leaf level the index structure links to the fragments providing an efficient access as shown in Figure 1. In principal, the proposed approach can be realized in three different ways: (1) Each fragment is stored as a separate relation, that is accessed using standard interfaces like ODBC, JDBC etc. (2) All fragments are stored in one relation and are accessed using the previously mentioned interfaces. (3) Alternatively, the Oracle Object Kernel API [Ora02a] can be used, allowing the storage of structured objects in an Oracle 9i lite DBMS.

For implementation purposes Oracle 9i lite is used as a DBMS on the mobile devices, as it is available for all operating systems of mobile devices [Ora02b]. Unfortunately, JDBC is not supported on all platforms, so it could not be used.

The following description of the implementation of the trie and the corresponding algorithms focuses on the linkage of cache entries in the tree, and advantages and disadvantages of the mentioned implementation alternatives.

The major advantage of the two ODBC approaches is that an already specified and standardized query language (SQL) can be used. As shown in Figure 4, the access to the cache entries is possible through the relation name. Hence, the trie node referencing a cache en-

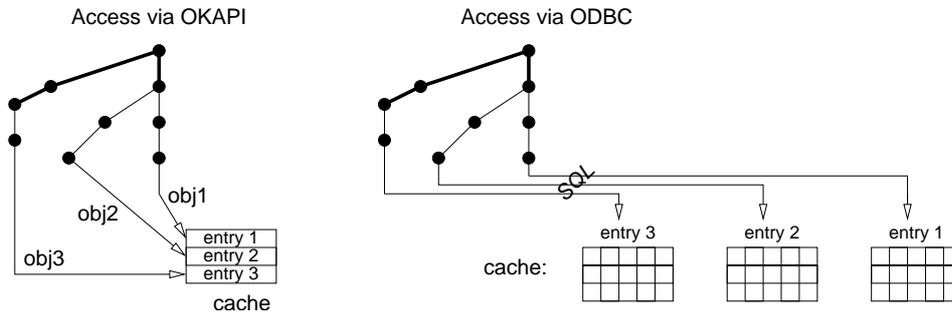


Figure 4: Cache entry referencing using OKAPI and ODBC

try contains a SQL query on the cached results. Unfortunately, this implies a performance loss. All queries, even those that can be answered from the cache, have to be performed via the ODBC interface. Furthermore, a separate transformation of the query given in calculus notation to SQL is required. Though this can easily be implemented, and is required for sending compensation queries to the server anyway, it implies an extra effort for local filter queries or if the result is included as such in the cache. If in this case the fragment is stored as a separate relation, the semantic context of the contained tuples is retained. If the results are all stored in one single relation, additional concepts for the identification have to be implemented to allow a fast filtering and retrieval of (partial) results⁶.

The third implementation alternative using the Object Kernel API of Oracle 9i lite works on a level below the SQL interpreter. Tables known in SQL are addressed as a group of objects in OKAPI. The function `okFindGroup` provides access to a group using its name. Accordingly, a trie node referencing a cached object, contains the path to the cache entry, as shown in Figure 4. The actual access to the cached results is implemented using an iterator object containing the search criteria. If the current query is equal to the cached query no search criteria is necessary. Otherwise, the search criteria is used as a filter over the cache fragment. The disadvantage of this approach is that the described additional query functionally has to be implemented.

A combination of the OKAPI and ODBC access is possible, according to [Ora02a], but in this application not necessary. The combination would become interesting, if there were further applications accessing the cache, requiring the SQL support of Oracle 9i.

5 Conclusion and Outlook

In this paper we presented an approach for local caching in a mobile database system environment. By providing trie-based indexing on the cache entries it is well suited for context based queries, which are typical in mobile information systems. We thoroughly discussed the trie index structure, and how new queries are “moved through” the trie to

⁶A trivial inefficient case is for instance the processing of an already cached query on the cache.

retrieve previously stored query results. In addition to these theoretical considerations and a comparison to related work, implementation alternatives based on Oracle 9i lite were described and compared.

Based on the presented work and the described implementation of the LDD cache the caching strategies introduced in Section 3 have to be evaluated and compared. Furthermore, the influence of certain query predicates has to be analyzed. Especially, the usage of cached data in join operations requires a more detailed specification and analysis. Based on the results the mobile cache will be integrated in the server infrastructure for context-based information systems presented in [HS03a]. Additional open issues are, for example:

- Ordering of predicates (cost-based)
- Splitting and merging of nodes
- Context-based cache replacement

Moreover, we currently adapt this data caching approach in order to handle a great number of mobile clients on a mobility support server. The idea is, that the cache index is used for indexing mobile clients. That means, that the indexed values represent the IDs of mobile clients and the index structure represents the queries which were given from these mobile clients. A first presentation of this idea can be found in [HS03b].

References

- [Bla98] Paul E. Black. NIST Dictionary of Algorithms and Data Structures. Web page, 1998. <http://www.nist.gov/dads/>.
- [DFJ⁺96] Shaul Dar, Michael J. Franklin, Björn THór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 330–341, Mumbai (Bombay), India, 3–6 September 1996. Morgan Kaufmann.
- [DK98] Margaret H. Dunham and Vijay Kumar. Location dependent data and its management in mobile databases. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, Vienna, Austria, August 24-28, 1998 / W07: Mobility in Databases and Distributed Systems*, pages 414–419. IEEE Computer Society, August 1998.
- [Fre59] E. Fredkin. *Trie memory*. Information Memorandum, Bolt Beranek and Newman Inc., Cambridge, MA, 1959.
- [GG99] Parke Godfrey and Jarek Gryz. Answering Queries by Semantic Caches. In *Database and Expert Systems Applications*, pages 485–498, 1999.
- [Gol03] Christoph Gollmick. Nutzerdefinierte Replikation zur Realisierung neuer mobiler Datenbankanwendungen. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Tagungsband der 10. Konferenz Datenbanksysteme für Business, Technologie und Web (BTW03)*, 26.-28. Februar 2003, Leipzig, volume 26 of *Lecture Notes in Informatics*. GI, 2003.
- [HS03a] Hagen Höpfner and Kai-Uwe Sattler. Semantic Replication in Mobile Federated Information Systems. In *Proceedings of the Fifth International Workshop on Engineering*

- Federated Information Systems (EFIS2003)*, Coventry, UK 17 - 18 July, 2003, pages 36–41. Akademische Verlagsgesellschaft Aka GmbH, Berlin, July 2003.
- [HS03b] Hagen Höpfner and Kai-Uwe Sattler. SMoS: A Scalable Mobility Server. In *Poster-Proceedings of Twentieth British National Conference on Databases (BNCOD20)*, Coventry, UK 15th - 17th July, 2003, pages 49–52. School of Mathematical and Informational Sciences; Coventry University, July 2003.
- [iAn03] iAnywhere Solutions, Inc. SQL Anywhere[®] Studio. Data sheet, Sybase, Inc., 2003. <http://www.sybase.com/content/1025129/SQL9Datashet.pdf>.
- [IBM02] IBM. IBM DB2 Everyplace Version 8.1. Spec sheet, IBM Corporation, 2002. <http://www-3.ibm.com/software/data/pubs/pdfs/db2epv8ss.pdf>.
- [IBM03] IBM. IBM Cloudscape 5.1: A Technical Overview. Whitepaper, International Business Machines, March 2003. <http://www-3.ibm.com/software/data/cloudscape/pubs/whitepapers/cloudscape-techover.pdf>.
- [LH01] Astrid Lubinski and Andreas Heuer. Configured Replication for Mobile Applications. In Janis Barzdins and Albertas Caplinskas, editors, *Databases and Information Systems: Fourth International Baltic Workshop, Baltic DB&IS 2000, Vilnius, Lithuania, May 1-5, 2000 Selected Papers*. Kluwer Academic Publishers, 2001.
- [Ora02a] Oracle. Oracle 9i lite Release 5.0.1: C and C++ Object Kernel Reference, 2002. http://otn.oracle.com/docs/products/lite/doc_library/release501/doc/okapi/html/toc.htm.
- [Ora02b] Oracle. Oracle 9i lite: Release Notes - Release 5.0.1, 2002. http://otn.oracle.com/docs/products/lite/doc_library/release501/readme.htm.
- [RD99] Qun Ren and Margaret H. Dunham. Using Clustering for Effective Management of a Semantic Cache in Mobile Computing. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access, August 20, 1999, Seattle, WA, USA*, pages 94–101. ACM Press, 1999.
- [RD00] Qun Ren and Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 210–221. ACM Press, 2000.