Journal of Computer Science 3 (10): 773-779, 2007
ISSN 1549-3636
© 2007 Science Publications

Query Based Client Indexing in Client/Server Information Systems

Hagen Höpfner
School of Information Technology, International University in Germany,
Campus 3, 76646, Bruchsal, Germany

Abstract: One issue in client/server information systems is the storage of the relationships between clients and data used by these clients. In particular in scenarios that allow the caching of data on the client site, this information can be used in order to keep the global database consistent. Thus, if the data on the server are updated, it is possible to detect caches affected by the update. In a following Step it is possible either to patch or to invalidate these caches. In this study we discuss approaches that use posted queries in order to index the clients on the server site.

Key words: Query index, Trie, stateful server, caching, client indexation

INTRODUCTION

Client/server information systems use caching techniques for reducing the volume of transmitted data. Data that have been received once are stored on the client and can be reused if the client requires parts of this data later on. In case of a server site update, this desired redundancy potentially leads, especially in loosely coupled systems like information systems with mobile clients, to inconsistencies within the global database. As shown in the following example, checking the relevance of such updates regarding the clients' caches has to be done on the server. Let us assume the following two relations:

Cinemas	CID	Name	Movie_ID	Time
	99	CinemaxX	01	4 pm
	99	CinemaxX	02	7 pm

Movies	MID	Title	Length
	02	Lord of the Rings III	210 min
	01	Matrix Reloaded	138 min

The client posted the query: SELECT name, title, time FROM cinemas, movies WHERE movie ID = MID AND title = 'Matrix Reloaded'. The result contains the following data:

Name	Title	Time
CinemaxX	Matrix Reloaded	4 pm

Therefore, the client is not able to decide locally about the relevance of a server site update like UPDATE cinemas SET time = '15:30' WHERE CID = 99 and movie ID = 01.

One possibility for avoiding this problem is to forbid the usage of operators (like the projection in the example) that remove attributes required for checking the update relevance. From the users point of view that means transmitting not required data. Therefore, this approach is not applicable. In a PhD-thesis[7] we have shown that it is better to check the relevance on the server. Therefore, it is necessary to know the state of each cache. Furthermore, one has to be able to assign caches to clients. In this study we discuss different approaches to realize such a client index that uses the queries posted by the clients and, therefore, contains the required cache descriptions as well as the cache client assignment. For illustration purposes we use a mobile information system. However, it is important to point out that our approach is applicable in any kind of loosely coupled information system.

The remainder of the study is structured as follows: First we give a brief overview of the query notation used to support the index by requiring a strict syntax. Then we discuss different indexing approaches and their evaluation. Finally, we summarize and conclude the study.

QUERY REPRESENTATION

Queries in mobile information systems are typically generated by an application. So, it is not necessary to support descriptive query languages like SQL. Instead, queries can be represented in a way that reduces the effort for converting them in order to be usable as index. The query notation used here corresponds to the well-known conjunctive queries with inequality comparisons but also supports self-join. It contains elements from relational algebra and from relational calculus. Queries are sequences of a set of selection predicates SP, a set of join predicates VP and a set of up to one projection predicate PP. The elements of each set are lexicographically ordered in a query. With $V \subseteq VP$, $pp \in PP \cup \{\varepsilon\}$, $S \subseteq SP$ and $V \cup \{pp\} \cup S \neq \emptyset$ a conjunctive query Q is represented as predicate sequence query (PSQ) $Q' = \langle vp_1 \ldots vp_n sp_1 \ldots sp_o pp \rangle$ with $i, k \in N$, $1 \leq i < k \leq n$; $vp_i, vp_k\ V \in \{\varepsilon\} \Rightarrow vp_i \blacktriangleleft vp_k$ and $i, k \in N$, $1 \leq i < k \leq o$; $sp_i, sp_k \in S \cup \{\varepsilon\} \Rightarrow sp_i \blacktriangleleft sp_k$. Here the symbol $\blacktriangleleft$ means lexicographically smaller.

The appearance of predicates in a PSQ is based on further conditions and self-joins are handled by renaming tables. Details about this query representation can be found in two previous publications[7, 9].

CLIENT INDEXING

As mentioned above, the aim of the client indexing is to be able to assign cache states to clients (and vice versa). If a client posts a query its ID is registered together with the query. So, the server is stateful. Therefore, in case of an update, it is possible to figure out which clients hold data affected by the update and to notify these clients.

In the following we use the cinema database schematically shown in Fig. 1. In contrast to the relations in the introduction this database supports cinemas with more than one auditorium.

Sequential storage of queries: Assigning queries to clients can be simplest done by using a two column relation (Table 1). The first column contains the queries and the second one contains a list of IDs of clients that posted the corresponding query. This easy solution benefits from its obvious compatibility to relational storage. Multi-valued attributes are not allowed in this data model. Therefore, both columns have to be strings or one has to remodel the schema. Due to the m:n-relationship (query/client) we would get three relations. As each entity type has only one attribute and due to the given cardinalities, we could reduce them to an relation Client Index (query,client). An alternative would be the usage of an object relational system, which allows multi-valued attributes.

Naive Trie-based indexing: Storing queries in a sequential manner is independent of query representation. However, in the following we discuss approaches that benefit from syntactical properties of PS-queries.

Prefix predicates that are common in various queries are, therefore, stored only once. Formally, PS-queries are words Q_i of the alphabet of the allowed predicates $P = PP \cup SP \cup VP$. Standard database textbooks suggest the usage of digital trees for searching in sets of words. One such an approach is the Trie[4] that stores the information by using the edges. Nodes include all letters of the alphabet and describe which edge has to use for completing a word. Let us, for example, assume that words might use all 26 letters of the English alphabet. A Trie would store the words data, dating and date as shown in Fig. 2. Remember, each node contains all 26 possible letters.

Obviously in case of a big alphabet, this approach leads very often to unused letters in the nodes (Fig. 2). To overcome the waste of memory one can use a Trie-approach that stores only needed letters. Figure 3 shows such a Trie for the three words mentioned above. It is based on de la Briandais' algorithm[3].

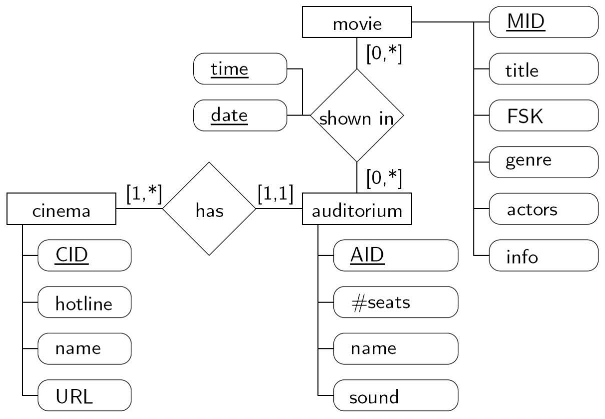

Fig. 1: ER diagram of the example database

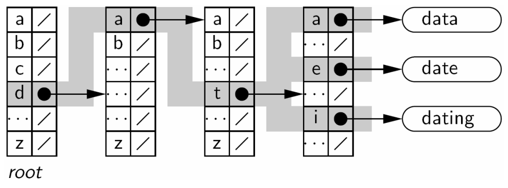

Fig. 2: Example of a standard Trie

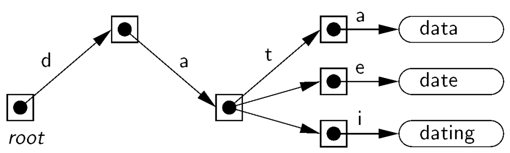

Fig. 3: Example of a memory saving Trie

Beside this approaches there are a few data structures like Compact Tries[16], Patricia trees[14,17] and Prefix trees[15] that reduce the depth of digital trees. The idea is to minimize the number of nodes but at the same time, to guarantee the indexing property. Compact Tries remove non-branching sub paths to leaf nodes. Patricia trees represent non-branching sub-paths as the number of involved edges only. Prefix trees are extended Patricia trees and store the skipped partial word (sub-path) in addition to the number. However, such compression techniques are not usable in our scenario because we have to guarantee the reconstruct-ability of the stored queries. Furthermore, we do not want to index the query string but the client IDs of clients having posted the particular query. Therefore, we adapt the original Trie. A query tree[10] is based on three sets of nodes: leaf nodes B, inner nodes I and the root node {root}. Each leaf node contains an ID-list of clients having posted the query represented by the path from the root node to this leaf node. The root node contains a nonempty list of references to inner nodes. Each inner node references either another inner node or a leaf node. The edges used as references between and to inner nodes are marked by a predicate. Edges to leaf nodes are marked with the empty predicate e.

As mentioned above it is not rational to store all letters in each node. Therefore, we implemented a query tree AB based on de la Briandais' Trie[3]. We already used a similar index structure for checking update relevancy[10]. For this reason we had to store temporary results TR and to represent the names of the relations used in the query separately. Furthermore, we extend the list of client-IDs and store it as CA = {(CID, AID)}, not separately, but in the last predicate node of a query. CID is a client-ID and AID is a query ID that is unique for each client. So, (CID, AID) is globally unique. A query tree is constructed of five node types:

- The root node root = $(0, K^c)$ with the node type ID 0 is the entrance point of a query tree and contains a list of links K^c to relation nodes
- A relation node $k_r = (1, \text{name}, K^c, k^p)$ with the node type ID 1 represents a relation name and contains a list of links K^c to relation nodes, join nodes, selection nodes, or projection nodes and a reverse link k^p to its parent node. Starting at root the relation nodes in a path are lexicographically ordered
- A join node $k^v = (2, vp, A_v, K^c, CA, k^p)$ with the node type ID 2 stores a join predicate $vp \in VP$. To support the update relevance check mentioned above, each last join node of a path contains a set A_v of attribute names used in predicates in the sub tree starting at this join node. The parent node of a join node might be a relation node or another join node. Child nodes might be projection nodes, selection nodes, or further join nodes
- A selection node $k_s = (3, sp, K^c, CA, k^p, TR)$ with the node type ID 3 stores a selection predicate $sp \in SP$ and contains a list of links K^c to projection nodes or selection nodes as
- well as a back link k^p to a selection node, a join node, or a relation node
- A projection node $k_p = (4, pp, k^p, CA, TR)$ with node type ID 4 stores a projection predicate $pp \in PP$ and contains a link back to a selection node, a join node or a relation node

Figure 4 shows the query tree for the queries of Table 1. Client IDs and query IDs are stored in the leaf nodes of the query tree. Hence, deregister a query requires a complete traversal. In order to improve this, we use a help index that allows bottom-up traversal of the tree. Therefore, we used a sorted list[6] and an AVL-tree[1,7]. This leaf node index (LNI) contains all client IDs and indexes the leaf nodes of the query tree in which a query of a particular client ends. We will not discuss this help structure in more detail here but assume that it exists.

Inserting and deleting a query: A client submits a query and its client ID (if available) in order to register a query. The server automatically assigns a new client ID to new clients. This client ID, a generated query ID and the query result are returned to the client.

The next step is to preprocess the query. Afterwards and without loss of generality, a query with r relation names, n join predicates, o selection predicates and one projection predicate is represented as an enhanced query EQ of the form $\langle(\text{name}_1, 1) \ldots (\text{name}_r, 1)(vp_1, 2) \ldots (vp_n, 2)(sp_1, 3) \ldots (sp_o, 3)(pp, 4)\rangle$. Therefore, names of relations and attributes are converted to uppercase, predicate types are computed and relation names are extracted. This enhanced query is then inserted into the query tree by using Algorithm 1.

Table 1: Naive sequential storage of queries

Query	Clients
⟨[movie, shown_in, (movie.MID = shown_in.MID)] [movie.FSK > 16] [movie.genre = 'action'] [movie(name), shown_in(time)]⟩	{23,66,20}
⟨[movie.FSK > = 18][movie(name, FSK, genre)]⟩	{200,11}
⟨[movie, shown_in, (movie.MID = shown_in.MID)] [movie(name), shown_in(date, time)]⟩	{66,200}
⟨[movie.FSK> = 18][movie(name, FSK)]⟩	{45,24}

Algorithm 1: Inserting a new query into the query tree

```
INPUT:    EQ      // preprocessed query
          CID     // client ID
          root    // root node of the AB
OUTPUT:   AID     // query ID
01        def find_last_equal_node(node, EQ, n)
02        if node.K^c ≠ ∅ ∧ n < |EQ|
03        for each child c ∈ node.K^c
04        let p be the n^th element of EQ
05        if c.value = = p.value ∧ c.type = = p.type
06        return (find_last_equal_node (c, EQ, n+1))
07        return (node, n)
08        return (node, n)
09
10        def insert_path(EQ, CID)
11        (node, pn) = find_last_equal_node (root, EQ, 0)
12        k = node
13        generate new AID by using LNI
14        if pn<|EQ|
15        insert the query suffix staring at predicate pn+1
16        let k be leaf node of the query path
17        else
18        if k referenced by CID in LNI
19        break
20        k.CA = k.CA ∪ {(CID, AID)}
21        Insert AID into LNI and link from (CID, AID) to k
22        if p_1 ∈ EQ ∧ p_2 ∈ EQ ∧ (p_1 ∈ VP ∧ p_2 ∈ SP ∧ PP)
23        Insert attribute names of selection predicates and projection predicates into the last join node
24        return (AID)
```

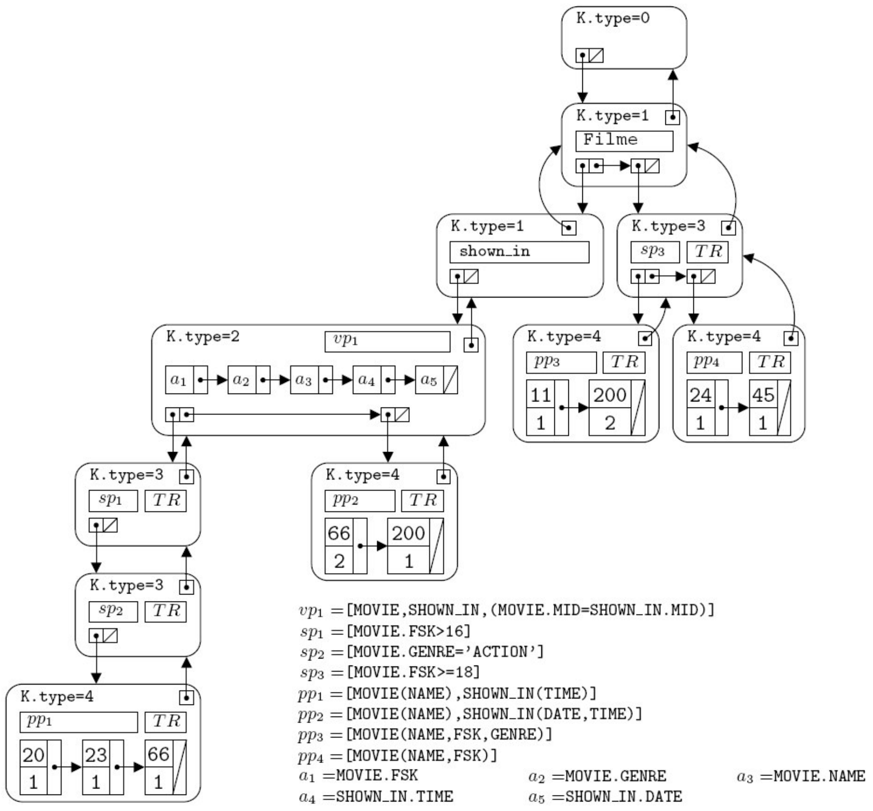

Fig. 4: Query tree for indexing client Ids

Algorithm 2: Removing a new query into the query tree

```
INPUT:  CID , AID // client ID and query ID
01      def delete_path(CID, AID)
02      let node be leaf node of the query path
03      node.CA = node.CA − {(CID, AID)}
04      free_nodes(node)
05      remove {(CID, AID)} from LNI
06
07      def free_nodes(node)
08      if node. CA = = ∅ ∧ node.K^c = = ∅
09      parent = node.k^p
10      parent.K^c = parent.K^c − node
11      free(node)
12      free_nodes(parent)
```

The function find_last_equal_node (line 01-08) computes the number n of reusable nodes and, thus, the prefix of the query that is already included in the tree. The remaining suffix is inserted into the tree (line 15). The next step is to generate (based on the LNI) a new query ID. If the given query was already completely contained in the tree, then we have to check whether this client has posted this query before (line 17-18). This is the case if there is a link from the LNI entry with the clients' ID to the leaf node k where the query ends. We then stop the insertion (line 19). In all other cases (query is at least partially new or was posted by a different client) we insert the query, register the new (CID, AID) in the LNI (line 20) and establish the link to the leaf node k (line 21). If the query contains join predicates and selection predicates or a projection predicate (line 22), then we have to insert the corresponding attribute names into the last join node (line 23).

If a client wants to deregister a query, then it submits its client ID and the corresponding query ID. However, deregistering a query might be a user driven process but can also be the result of a cache replacement decision. In this study we do not discuss the client implementation but can point out that our approach is applicable to any kind of client/server information system that use, for example, caching, hoarding, or replication. With help of the LNI deregistering is done bottom-up. At first we request the leaf note k of the query from LNI and remove (CID, AID) from the CA-list in k. Now we can remove k if the CA-list is empty (no other query has posted this query). The next step is to look at k's parent node. If its CA-list is empty and k was the only node, then we can remove this node too and so on. Algorithm 2 realizes this procedure.

Beside this query tree, we implemented an optimized query tree OAB. Because of the given space limitation we were not able to include this idea in this study but refer to the original German paper[8] that includes the algorithms as pseudo code. However, the idea is to use the commutability of selection predicates in order to reduce the number of nodes in a query tree.

Table 2: Number of predicates per query set

	Set 1	Set 2	Set 3	Set 4
VP	1579	3792	6017	3948
	(26)	(42)	(45)	(45)
SP	14530	24978	41709	27584
	(2287)	(2462)	(2708)	(2528)
PP	8799	9343	9481	9170
	(390)	(785)	(1275)	(896)

EVALUATION

The algorithms discussed in this study were implemented in the programming language Python (version 2.2.3). We used a standard PC with an AMD AthlonTMXP 2000+ processor (1666.663 MHz) and 512 MB RAM running SuSE Linux 9.1 (SuSE specific kernel 2.6.5). We do not discuss the results of the sequential indexing but focus on the query tree. For the evaluation we created four sets of queries:

Set 1: queries with 1 to 3 predicates (short queries)
Set 2: queries with 3 to 5 predicates (mid length queries)
Set 3: queries with 5 to 7 predicates (long queries)
Set 4: queries with 1 to 13 predicates

The query generator worked in two steps. At first we generated 60000 candidate queries from which we selected the aforementioned four duplicate free sets. Table 2 illustrates the usage of the different predicate types within the query sets. The number in brackets stands for the number of different predicates of the same type used in the predicate set. The other number stands for the total number of predicates at this time in the particular set.

Space consumptions: Due to the characteristics of the programming language used, it is nearly impossible to evaluate the space consumptions of the query tree. However, we can point out, that the trees of all four query sets fit into main memory. The Python process used maximally 8% of the available memory. Instead of discussing a Kilobyte number here, we use the number of nodes as evaluation criterion. Query set 1 contains 24908 predicates that are represented by 15662 nodes in the tree. The ratio of query set 2 was 38113 predicates to 25436 nodes. For query set 3 that contains 57207 predicates, 41277 nodes were required and the 40702 predicates of query set 4 resulted in 28840 nodes. If we look at the procedural values (set 1: 37.12%, set 2: 33.26%, set 3: 28%, set 4: 29.14%) it becomes obvious that shorter queries benefit more than longer ones from the tree representation. The reason is that shorter queries contain fewer predicates that lead to less variety. Hence, the possibility to find two syntactically overlapping queries is higher for shorter queries.

The optimized query tree that was mentioned at the end of previous Section reduces the number of required nodes further. Set 1 required 15517 nodes, set two 24171 nodes, set three 38276 nodes and set four 24742 nodes. Compared to the not optimized query tree this means a reduction by 1% for set 1, 5% for set 2, 7% for set 3 and 14% for set 4. Obviously, longer queries benefit from the optimization that simply rearranges sub-trees that consist of selection predicates only in such a way that the number of required nodes is minimized. Since longer queries probably contain more selection predicates, this optimization is more suitable for longer queries.

Time consumption for inserting queries: Inserting a query into the query tree is comparable to inserting a word into a Trie. Due to the list implementation of child node links within a parent node, it is not possible to reach a constant time complexity that is theoretically possible for equally long words. Depending on the length of these lists our algorithm has to check different numbers of child nodes.

As illustrated in Fig. 5, the length of a query has a small impact on the time required for insertion. However, as illustrated in Fig. 6 one can neglect this issue. The outliers resulted from background activities of the test computer. They are only recognizable because inserting a query took less then 0.02 seconds.

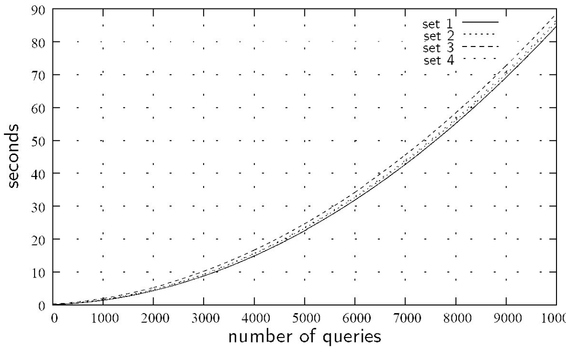

Fig. 5: Insert into tree-cumulative time

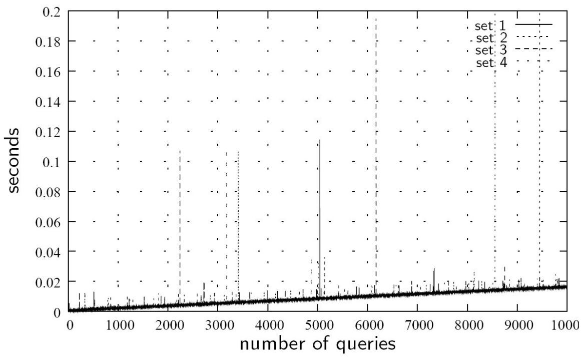

Fig. 6: Insert into tree-time per query

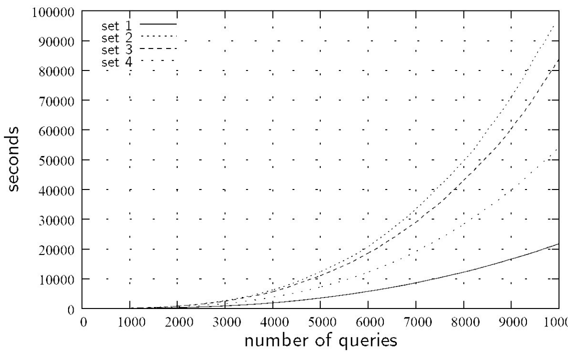

Fig. 7: Optimized insertion-cumulative time

The algorithm briefly mentioned before for inserting queries into an optimized query tree is more complex than the not optimized version.

We also discussed the issue that the optimization reduces the number of nodes that are required for representing all queries. However, as shown in Fig. 7, the optimization takes a lot of time. Since we did not discuss the algorithm in detail in this study we also do not discuss its evaluation here.

CONCLUSION

In this study we discussed and evaluated data structures for realizing a stateful database server that indexes clients' cache states and assigns them to the clients. The results show that the idea of digital trees is adaptable to query indexes and reduces the number of predicates/nodes required for storing the queries. Looking at an update relevance check, that was not part of this study, we can point out that a lower number of nodes also reduce the time required for checking the relevance of a server site update. With an optimized insert algorithm, which suffers from its time consumptions, we can reduce the number of nodes further.

This study is part of ongoing work. One of the next steps will be to analyze the time complexity of our algorithms in more detail and to reduce the time consumption of the optimized insert function. Furthermore, we have to investigate whether these time consumptions are due to the used implementation language or whether they result from bad time complexity of the algorithm. Independent of this analysis it seems to be a good idea to combine the normal insert with the optimization in such a way, that the tree is built up normally but optimized from time to time.

REFERENCES

1. Adelson-Velskii, G.M. and E.M. Landis, 1962. An Algorithm for the Organization of Information. Soviet Math. Doklady, 3: 1259-1263.
2. Beckmann, N., H.-M. Kriegel, R. Schneider and B. Seeger, 1990. The R-tree: An Efficient and Robust Access Method for Points and Rectangles. ACM SIGMOD Record, 19: 322-331.
3. de la Briandais, R., 1959. File Searching Using Variable Length Keys. In: Proceedings of the AFIPS Western Joint Computer Conference, AFIPS Press, Montwale, pp: 295-298.
4. Fredkin, E., 1959. Trie Memory. Information Memorandum, Bolt Beranek and NewMan Inc., Cambridge.
5. Guttman, A., 1984. R-trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD Record, 14: 47-57.
6. Höpfner, H., 2004. Serverseitige Auswertung von Indexen semantischer, clientseitiger Caches in mobilen Informationssystemen. Proceedings der 34. GI Jahrestagung (P. Dadam and M. Reichert, Ed.) Köllen Druck+Verlag GmbH, Bonn. (in German), pp: 298-302.
7. Höpfner, H., 2005. Relevanz von Änderungen für Datenbestände mobiler Clients. Fakultät für Informatik der Otto-von-Guericke Universität Magdeburg. (in German).
8. Höpfner, H., 2006. Anfragebasierte Client-Indexierung in Client-Server-Informations systemen. Informatik-Forschung und Entwicklung (in German), 20: 209-221.
9. Höpfner, H. and K.-U. Sattler., 2003. Towards Trie-Based Query Caching in Mobile DBS. In: Persistence, Scalability, Transactions-Database Mechanisms for Mobile Applications (B. König-Ries, M. Klein and P. Obreiter, Ed.) Köllen Druck+Verlag GmbH, Bonn, pp: 106-121.
10. Höpfner, H., S. Schosser and K.-U. Sattler, 2004. An Indexing Scheme for Update Notification in Large Mobile Information Systems. In: Current Trends in Database Technology-EDBT 2004 Workshops (W. Lindner, M. Mesiti, C. T¨urker, Y. Tzikzikas and A. Vakali, Ed.), Springer-Verlag, Heidelberg/Berlin, pp: 345-354.
11. Hagen Höpfner, Can Türker and Birgitta König-Ries, 2005. Mobile Datenbanken und Informationssysteme-Konzepte und Techniken. dpunkt.verlag, Heidelber., (in German).
12. Lee, M.L., W. Hsu, C.S. Jensen, B. Cui and K.L. Teo, 2003. Supporting Frequent Updates in R-trees: A Bottom-up Approach. In: Proceedings of the 29th Conference on Very Lage Databases (J.-C. Freytag, P.C. Lockemann, S. Abiteboul, M. Carey, P. Selinger and A. Heuer, Ed.), Morgan Kaufmann Publishers Inc., San Fransisco, pp: 608-619.
13. Lehner, W., 2002. Subskriptionssysteme-Marktplatz für omnipräsente Informationssysteme. B.G. Teubner GmbH, Stuttgart. (in German).
14. Morrison, D.R., 1968. PATRICIA-Practical Algorithm to Retrieve Information Coded in Alphanumeric. Journal of the ACM (JACM), 15: 514-534.
15. Gunter Saake, Andreas Heuer and Kai-Uwe Sattler, 2005. Datenbanken: Implementierungstechnicken MITP-Verlag GmbH, Bonn. (in German).
16. Sussenguth, E.H. Jr., 1963. Use of tree Structures for Processing Files. Communications of the ACM, 6: 272-279.
17. Szpankowski, W., 1990. Patricia Tries Again Revisited. Journal of the ACM (JACM), 37: 691-711.