Semantic Caching and Query Processing

Article *in* IEEE Transactions on Knowledge and Data Engineering · February 2003

DOI: 10.1109/TKDE.2003.1161590 · Source: IEEE Xplore

Margaret Dunham
Southern Methodist University
101 PUBLICATIONS **2,340** CITATIONS
SEE PROFILE

Vijay Kumar
University of Missouri–Kansas City
103 PUBLICATIONS **1,597** CITATIONS
SEE PROFILE

Semantic Caching and Query Processing

Qun Ren, Margaret H. Dunham, and Vijay Kumar, *Member*, *IEEE*

Abstract—Semantic caching is very attractive for use in distributed systems due to the reduced network traffic and the improved response time. It is particularly efficient for a mobile computing environment, where the bandwidth of wireless links is a major performance bottleneck. Previous work either does not provide a formal semantic caching model, or lacks efficient query processing strategies. This paper extends the existing research in three ways: formal definitions associated with semantic caching are presented, query processing strategies are investigated and, finally, the performance of the semantic cache model is examined through a detailed simulation study.

Index Terms—Semantic caching, query Processing, query trimming, client-server, mobile computing, page caching.

1 Introduction

Caching of frequently accessed data at clients plays an important role in improving the performance and achieving system scalability in distributed systems ([6], [9]). Page caching is a traditional approach and is widely used in client-server systems. It intends to exploit static *spatial* and/or temporal locality and, hence, has inherent disadvantages. The performance of page caching is very sensitive to the database physical organization, a low degree of spatial locality can cause high network communication and space overheads. Also, since no semantic information about cached items is kept, the client highly depends on the network connection to consult with the server about how the local cache matches with query results. Obviously, this is not an ideal solution for mobile computing, where the network connection is not always reliable.

The idea of *semantic caching* is that the client maintains in the cache both semantic descriptions and results of previous queries ([5], [17]). If a new query is totally answerable from the cache, no communication with the server is necessary; if it can only be partially answered, the original query is trimmed and the trimmed part is sent to the server and processed there. By processing queries in this way, the amount of data transferred over the network can be substantially reduced. To further illustrate the advantages of semantic caching, we compare it with page caching in Table 1.

Since semantics about the cached items are stored in a semantic cache, the client is able to reason from the local cache to determine whether a query can be totally answered, how much it can be answered, and what data are missing. Hence, the network communication cost is reduced, as only the required data are sent from the server to the client. On the other hand, in a page caching scheme, the unit of transfer over the network is a page. When the database exhibits a poor clustering degree, only a small portion of tuples in a transferred page may be useful. Additionally, it is very difficult for a page caching client to judge whether the entire result of a query could be obtained locally. It has to contact the server for possibly extra data items. In a word, semantic caching is expected to be more effective than page caching in saving communication cost, which is a very important advantage in mobile environments. For similar reasons, semantic caching is expected to make better use of cache space than page caching in many applications. Both schemes involve space overheads for cache management, such as index in semantic caching and buffer control blocks in page caching. The performance, in terms of space overhead, will be more convincely compared in the later simulation study.

Through reasoning among the query and the cached semantic descriptions, the exact work to be done at the server side can be determined beforehand, hence we can let the client and the server work in parallel. By doing so, the client side processing overhead can be partially hidden and the system performance as well as the response time is expected to be greatly improved. However, it is very difficult to realize such parallelism in a page caching scheme. In general, page caching is faulting-based ([5]). The missing pages are not predetermined, instead, they are requested from the server during the query processing procedure.

At last, the self-reasoning ability of semantic caching makes it an effective scheme for mobile computing. Mobile clients can become more autonomous. When disconnection occurs, mobile clients are still able to determine whether a query can be answered locally or not, even though the server is not available.

From what we have discussed so far, it can be seen that semantic caching exhibits many advantages over page caching. There has been much recent interest in this area. Previous work either ignores the formal definitions, or lacks precise query processing algorithms. Some of them only work in specific application domains, such as [8] and [2], while others limit query types, such as [5] which focus on

- *Q. Ren and M.H. Dunham are with the Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275. E-mail: qun.ren@nokia.com, mhd@engr.smu.edu.*
- *V. Kumar is with Computer Science Telecommunications, University of Missouri-Kansas City, Kansas City, MO 64110. E-mail: kumar@cstp.umkc.edu.*

Manuscript received 23 Feb. 2000; revised 16 Oct. 2000; accepted 20 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 111546.

TABLE 1
Semantic Caching versus Page Caching

Item	SC	PC	Reason
communication cost	low	high	only required data transferred & compact expression of request in SC
cache space overhead	low	high	only data satisfying previous queries are stored in SC
parallelism in query processing	easy	difficult	missing data are exactly determined as SC keeps semantic information
used in mobile computing	efficient	inefficient	more autonomy of mobile clients & partial results derived in SC when disconnected

only select operations. The objective of this paper is to extend the existing semantic caching work along several dimensions. First, we work on select-project queries in general application domain and present a formal semantic caching model. Then, we explore the semantic caching query processing strategies. After dealing with processing a query via a single cached segment, we further examine how to efficiently answer queries against the whole cache. Finally, we validate semantic cache performance through a detailed simulation study.

The rest of this paper is organized as follows: In Section 2, we review previous research on semantic caching as well as other related issues. A formal semantic cache model is defined in Section 3. Section 4 discusses semantic caching query processing strategies. The performance of semantic caching is examined through a simulation study in Section 5. Finally, we summarize our work and discuss future research in Section 6.

2 Related Work

Semantic caching has been widely studied in previous literature where applications of this technique in different areas can be found. In centralized systems, previous query results are cached in memory to save the subsequent query computations and generate more efficient query plans ([23], [3]). While it is mainly used to reduce disk access in a centralized system, the semantic caching scheme in a client-server environment aims at alleviating the network contention and improving the system scalability ([5], [17]). Moreover, semantic caching is particularly attractive for use in mobile computing due to the fact of more autonomy of the mobile clients ([19]). Applications in specific domains such as OLAP ([8]), LDAP ([2]), WWW ([18]), and heterogeneous systems ([10]), are also explored. In what follows, we briefly review previous semantic caching research from two aspects: cache model and query processing.

2.1 Cache Model

In [17], Keller and Basu call their cache a predicate-based cache. Data are loaded into the client cache based on the queries executed. They allow general select-project-join queries over one or more relations. [17] does not specify cached items individually, instead, it treats the cache as a whole and describes it using a collection of cached query predicates. Also, it does not examine how to organize and store the cached items. The semantic cache proposed in [5] is also based on the queries executed. Unlike [17], they specify cached items individually. The semantic cache is defined as a collection of *semantic regions*, each having a constraint formula describing its content. [5] considers only select queries. Other semantic caching approaches have been proposed elsewhere, with [19] in mobile computing, [8] in OLAP, [10] in heterogeneous systems, etc. However, most of the semantic cache papers to date have not formally defined the queries they use, the cached items and the cache itself. We believe that a formal model is the basis for further research, hence, we provide one in this paper.

A query result can be cached as actual data called a *materialized view* ([5], [19]), or as a number of pointers pointing to records of underlying relations needed to compute it ([23], [3]). Compared to the data based approach, the pointer based approach is more space-effective and more versatile, since it provides a way to access all attributes of base relations instead of only projected attributes ([3]). However, the pointer approach is not profitable in a distributed system, especially in a low bandwidth mobile computing environment. [5] adopts the data based approach, as it only deals with select queries, sufficient information can be kept for future query processing. Select-project queries are considered in [19] with project attributes automatically concatenated with the key attribute(s). This approach combines both the advantages of data and pointer based approach, in which key attributes act as a clue for those missing but useful attributes.

Another aspect of semantic cache organization is how to partition and coalesce the corresponding disjoint subregions (fragments) after a query processing. Always coalescing strategy results in inefficient cache space utilization since a coarse cache granularity is not effective in capturing frequently referenced data. On the other hand, the never coalescing approach leads to a large number of semantic segments, which may bring excessive overhead for query processing. The strategy used in [5] is to coalesce regions with the same cache replacement value into one region. Because they deal with only selections, this approach will not cause any difficulties in expressing the semantic meaning of the resulting region. [19] solves the coalescing problem by proposing an additional lazy partitioning strategy called *logical cache partitioning* as well as the straightforward *physical cache partitioning* which always partitions fragments eagerly. In this paper, we aim to develop coalescing strategies that are effective in both cache management and query processing.

2.2 Query Processing

Much work has been done in processing queries from individual cached items, however, how to compute queries using the whole cache has not yet drawn much attention.

The concept of *query trimming* first introduced in [17] is the procedure to split a query into two disjoint pieces when it partially overlaps with a cached semantic item: the overlapped part and what is left. A similar idea is also proposed in [5], which further calls the two split parts *probe query* and *remainder query*, respectively. However, no detail query processing strategies are given in either work. The work in [10] and [19] categorize the possible relationship between a query and a cached fragment into several different scenarios. For every scenario, [19] gives corresponding query processing strategies, but it does not provide the related theoretic base. The work in [2] formally considers the *cache-answerability* problem and also provides algorithms, yet it works for LDAP (Lightweight Directory Access Protocol) queries which are different from select-project queries in relational databases. In [8], a query is first analyzed in terms of a list of required *chunks*, then, according to this list, it gets split into two parts. One part can be answered from the cache, the other part, consisting of the missing chunks, has to be computed from the backend. This approach works well in OLAP systems, rather than for general applications.

Checking the satisfiability or implication relationship between a query and a semantic fragment is the first step for query processing. Given a query predicate Q and a fragment predicate P, there are three scenarios:

- $Q \Rightarrow P$, i.e., Q implies P, implying that the whole answer of Q is contained in P;
- $Q \wedge P$ is satisfiable, implying that part of Q's answer is contained in P;
- $Q \wedge P$ is unsatisfiable, implying that there is no common part between Q and P.

A lot of works have been done in solving the satisfiability and implication problems ([22], [25], [24], [14], [15]). Efficient algorithms are proposed for certain classes of queries such as conjunctive queries involving arithmetic inequalities. Previous works also indicate that when the $\neq$ comparison between variables is allowed, the problems become more difficult, or even NP-hard in the integer domain ([22], [24]). Even when $Q \Rightarrow P$, Q may not be computed from P. This situation happens when either P lacks of the attributes that occur in Q's predicate for further qualifying the result, or some of Q's attributes are not contained in P. Hence, to solve the computable problem, not only the predicate relationship but also the relationship between attributes must be examined. The related computable work can be found in [20] and [2].

Processing queries via a semantic cache is examined in [5], [3], and [17]. The work in [5] works on only select queries, it treats all the candidate semantic regions as a whole to compute the corresponding probe and remainder queries. The work in [3] works in a centralized system model and it mainly deals with optimization problems. A brief discussion is presented in [17] without any detailed strategies.

Our semantic caching query processing work differs from the previous research in three significant aspects. First, we work on select-project queries. Second, we examine the strategies of processing queries via a single cached item as well as from the whole cache. Third, we further optimize the query processing procedure to parallelize the work between the server and the clients.

3 Semantic Caching Model

We consider select and project queries in this research, although the proposed model can be easily extended to handle other more complicated queries. The formal definition of semantic caching and related concepts are first given in this section, then we discuss how to store and organize a semantic cache.

3.1 Formal Definition

In relational databases, a *relation* consists of a *relation schema* and a *relation instance*. The schema of a relation specifies its name and the name and type of each attribute. An instance of a relation is a set of tuples, in which each tuple has the same number of attributes as the relation schema. A *relational database* is a collection of relations with distinct relation names. The *relational database schema* is the set of schemas of the relations in the database. An *instance* of a relation database is a collection of relation instances, one per relation schema in the database schema ([21]). In this paper, when there is no risk of confusion, we use "relation" ("database") to refer the instance of "relation" ("database").

Suppose that the database examined, D, consists of a number of base relations R_1, R_2, ..., R_n, i.e., D = $\{R_i, 1 \leq i \leq n\}$. Further, let A_{R_i} stand for the attribute set defined by the schema of R_i, and A represent the attribute set of the whole database, then we have $A = \cup A_{R_i}, 1 \leq i \leq n$. Before defining a semantic cache, we first give the definition of predicates, through which a semantic cache is constructed.

Definition 1. *Given a database $D = \{R_i\}$ and its attribute set $A = \cup A_{R_i}, 1 \leq i \leq n$, a* **Compare Predicate** *of D, P, is of the form P = a op c, where $a \in A, op \in \{\leq, <, \geq, >, =\}$, c is a constant in a specific domain.*

As discussed in Section 2.2, the comparison operator $\neq$ brings about complexity in reasoning among predicates, the problem even becomes NP-hard in the domain of integers. Since we are more concerned about how to develop efficient query processing strategies for semantic caching, without a loss of generality, we ignore $\neq$ comparison in this study.

A semantic cache is composed of a set of cached items which we define as *Semantic Segments*. A semantic segment is an original, decomposed, or coalesced query result. Its definition is consistent with that of a materialized view ([11]). To further simplify the problem, in this study, we assume that the selection condition of a query is an arbitrary constraint formula of compare predicates, namely, a disjunction of conjunctions of compare predicates.

Definition 2. *Given a database $D = \{R_i\}$ and its attribute set $A = \cup A_{R_i}, 1 \leq i \leq n$, a* **Semantic Segment,** *S, is a tuple $< S_R, S_A, S_P, S_C >$, where $S_C = \pi_{S_A}\sigma_{S_P}(S_R)$, $S_R \in D$, $S_A \subseteq A_{S_R}$, and $S_P = P_1 \vee P_2 \vee ... \vee P_m$ where each P_j is a conjunctive of compare predicates, i.e.,*

$$P_j = b_{j1} \wedge b_{j2} \wedge \ldots \wedge b_{jl},$$

TABLE 2
Example of Semantic Cache Index

S	S_R	S_A	S_P	S_C	S_{TS}
S_1	Employee	{Ename}	30 < Age < 40	2	T_1
S_2	Project	{Pname, Budget}	Budget > 10k	4	T_2
S_3	Employee	{Ename, Salary}	Age ≤ 28	3	T_3
S_4	Employee	{Ename, Age}	Age > 45	8	T_4

each b_{jt} is a compare predicate involving only the attributes in A_{S_R}.

In Definition 2, S_R and S_A define the relation and attributes involved in computing S, respectively, S_P indicates the select condition that the tuples in S satisfy. In a word, these three elements specify the semantic information associated with S. The actual content of S is represented by S_C. From the restrictions added, we can see that semantic segments are the results of Select-Project operations, with the selection conditions containing only compare predicates.

Since semantic segments are actually cached query results, we can qualify the semantic information of queries in the same way as we specify those for segments. However, before queries get answered, their contents are empty (i.e., $Q_C = \phi$). Therefore, we formally define a query just as we define a segment.

Definition 3. *A* **Query** *Q is a semantic segment, $< Q_R, Q_A, Q_P, Q_C >$.*

A semantic cache is defined as a set of semantic segments. To reduce space overhead, the cached segments do not overlap with each other. In the following, we first give the concept of *disjointed* segments, then formally define a semantic cache.

Definition 4. *Semantic segments $S_i =< S_{i_R}, S_{i_A}, S_{i_P}, S_{i_C} >$ and $S_j =< S_{j_R}, S_{j_A}, S_{j_P}, S_{j_C} >$ are said to be* **disjointed** *if and only if*

1. *$S_{i_A} \cap S_{j_A} = \Phi$ or*
2. *$S_{i_P} \wedge S_{j_P}$ is unsatisfiable.*

Definition 5. *A* **Semantic Cache***, C, is defined as $C = \{\textit{semantic segment } S_i\}$, where ∀ j, k*

$$(S_j \in C \wedge S_k \in C \wedge j \neq k \Rightarrow$$

S_j and S_k are disjointed).

3.2 Semantic Cache Organization

There are several different ways to physically store semantic segments. The work in [5] stores segments in tuples, and associates every segment with a pointer to a linked list of the corresponding tuples. This approach works fine for select-only queries and memory caching. The key advantage is easy maintenance: tuples can be added, deleted, or moved between segments conveniently. However, this linked list scheme is not appropriate for disk caching, since it may result in too many I/O operations. Moreover, when select-project queries are cached, the resulting tuples for different segments are no longer at the same length. Hence, even for memory caching, its advantage in maintenance is lost. Another noticeable disadvantage for this approach is the large space overhead caused by the tuple pointers.

In this research, a semantic cache is composed of two parts: the content and the index. Every semantic segment is stored in one or multiple linked pages, and is associated with a pointer pointing to its first page in the memory (disk) cache. Notice that each page now contains query results, rather than database relations. The cache space is also managed at a page level, which makes segment allocation and deallocation algorithms more straightforward and simpler. For allocation, if there are enough free pages to hold a segment, then allocate the pages to it; for deallocation, just mark the deallocated pages as free. The index part maintains the semantic as well as physical storage information for every cached segment. In what follows, we list the basic items kept in the index. For every cached segment, we have:

- the name S, the relation S_R, the attribute set S_A, and the selection predicate S_P;
- the pointer pointing to the first page that stores the segment; and
- the timestamp indicating when the segment was last visited S_{TS};

Obviously, this index structure is consistent with the formal definition of the semantic cache. In addition to the four basic components of a segment, we further add other items for maintenance use, such as S_{TS}. The semantic cache index is more clearly illustrated through the following Example 1.

Example 1. Consider a project management database with two relations: Employee (Eno, Ename, Age, Salary) and Project (Pno, Pname, Eno, Budget); also suppose that the cache contains four segments:

- S_1: **Select** Ename **From** Employee **Where** $30 < Age < 40$;
- S_2: **Select** Pname, Budget **From** Project **Where** $Budget > 10k$;
- S_3: **Select** Ename, Salary **From** Employee **Where** $Age \leq 28$;
- S_4: **Select** Ename, Age **From** Employee **Where** $Age > 45$;

Also, suppose that the first pages of the four segments are two, four, three, and eight, respectively, and S_i was last visited at T_i, then the index is shown as Table 2.

4 Semantic Caching Query Processing

To process a query from a semantic cache, we first check whether it can be answered by the cache. If yes, the locally available results are computed directly from the cache. When the query can only be partially answered, we trim the original query by removing or annotating the already answered parts and send it to the database server for processing. In this section, algorithms for semantic caching query processing are examined.

4.1 Theoretic Foundation

Definition 6. *Consider a semantic segment*

$$S = < S_R, S_A, S_P, S_C >$$

and a query $Q = < Q_R, Q_A, Q_P, Q_C >$, *we say Q is* **answerable** *from S, if there exist a relational algebra expression F containing only project and select operations, and only involving attributes in* S_A, *such that* $F(S_C) \neq \phi$, *and* $\forall t\ (t \in F(S_C) \Rightarrow\ (t \text{ satisfies } Q_P \wedge\ t \text{ contains only attributes in } Q_A))$. *Furthermore, if* $F(S_C) = Q_C$, *we say Q is* **fully answered** *from S; otherwise, we say Q is* **partially answered** *from S.*

Definition 6 is derived from the concept of *Derivability* defined in [20]. From Definition 6, we know that the key to compute a query from a cached segment is to find the function F, and to make sure that F can be executed on the segment. Sometimes, even the entire result of a query Q is contained in a segment S, Q still is not answerable from S. This is because some of the attributes needed in F can not be found in S. So, in Definition 6, we add an additional restriction on F. The following Example 2 illustrates such a point.

Example 2. Consider the project management database and the semantic cache described in Example 1, suppose there comes a query Q = **Select** Ename **From** Employee **Where** $(Age > 50) \wedge (Salary > 50k)$. Obviously, the result of Q is totally contained in S_4, since every tuple which satisfies $(Age > 50) \wedge (Salary > 50k)$ will always satisfy $(Age > 45)$. However, Q can not be computed from S_4, because we cannot find an F as specified in Definition 6. Intuitively, Q seems to be computed from S_4 by a function $\pi_{Ename}\sigma_{(Age>50)\wedge(salary>50k)}$. But the attribute "Salary" used in this function is not in S_4 after the projection.

Definition 7. *Consider a query* $Q = < Q_R, Q_A, Q_P, Q_C >$, *the* **predicate attribute set,** Q_{P_A}, *contains all the attributes that occur in* Q_P, *i.e.,*

$$Q_{P_A} = \{a \mid a \text{ is an attribute, and } a \text{ occurs in } Q_P\}.$$

Lemma 1. *Consider a semantic segment* $S = < S_R, S_A, S_P, S_C >$, *a query* $Q = < Q_R, Q_A, Q_P, Q_C >$, *and Q's predicate attribute set* Q_{P_A}. *Then we have:*

1. *If* $S_R = Q_R$, $S_A \cap Q_A \neq \phi$, $Q_P \wedge S_P$ *is satisfiable by* S_R, *and* $Q_{P_A} \subseteq S_A$, *then, Q is answerable from S.*
2. *If* $S_R = Q_R$, $Q_A \subseteq S_A$, $Q_P \Rightarrow S_P$, *and* $Q_{P_A} \subseteq S_A$, *then, Q is fully answered from S.*

Proof. Let us first prove Statement 1. We construct a new segment $S' = < S_R, Q_A \cap S_A, Q_P \wedge S_P, S_C{}' >$. From Definition 2, we have $S_C{}' = \pi_{Q_A \cap S_A}\sigma_{Q_P \wedge S_P}(S_R)$. According to the relational algebra equivalence rules ([21]), we have:

- $S_C{}' = \pi_{Q_A \cap S_A}\sigma_{Q_P \wedge S_P}(S_R) = \pi_{Q_A \cap S_A}(\pi_{S_A}\sigma_{Q_P \wedge S_P}(S_R)) = \pi_{Q_A \cap S_A}(\pi_{S_A}\sigma_{Q_P}(\sigma_{S_P}(S_R)))$
  Since $Q_{P_A} \subseteq S_A$, then
- $S_C{}' = \pi_{Q_A \cap S_A}(\pi_{S_A}\sigma_{Q_P}(\sigma_{S_P}(S_R))) = \pi_{Q_A \cap S_A}\sigma_{Q_P}(\pi_{S_A}\sigma_{S_P}(S_R)) = \pi_{Q_A \cap S_A}\ \sigma_{Q_P}(S_C)$.

Let $F = \pi_{Q_A \cap S_A}\sigma_{Q_P}$. Since $Q_A \cap S_A \neq \phi, Q_P \wedge S_P$ is satisfiable, we have $S_C{}' = F(S_C) \neq \phi$. Moreover, because $Q_{P_A} \subseteq S_A$, we know all the attributes involved in F are contained in S. Since $S_R = Q_R$, from the definition of S', we know that each tuple t in $S_C{}'$ satisfies $Q_P \wedge S_P$, thus, t must satisfy Q_P. Moreover, $Q_A \cap S_A \subseteq Q_A$, hence, t contains only attributes in Q_A. According to Definition 6, we know that Q is answerable from S.

Now, we prove Statement 2. We construct the same new segment $S' = < S_R, Q_A \cap S_A, Q_P \wedge S_P, S_C{}' >$ as above. Similarly, we have $S_C{}' = \pi_{Q_A \cap S_A}\sigma_{Q_P}(S_C)$, and $F = \pi_{Q_A \cap S_A}\sigma_{Q_P}$, which involves only the attributes contained in S. Since $S_R = Q_R$, $Q_A \subseteq S_A$, then, $Q_A \cap S_A = Q_A$; further because $Q_P \Rightarrow S_P$, we have $Q_P \wedge S_P = Q_P$. Therefore, $S_C{}' = Q_C$, i.e., $F(S_C) = Q_C$. According to Definition 6, we know that Q is fully answered from S. □

Obviously, the conditions required for checking answerability in Lemma 1 are very strict. In the following, we examine other alternatives to reason among queries and segments. Definition 8 is derived from [20], which aims at examining the relationship among attributes.

Definition 8. *Let P be a predicate with attributes*

$$a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_m.$$

The attribute b_i, $1 \leq i \leq m$, *is said to be* **uniquely determined** *by* $a_1, a_2, \ldots a_n$, *with respect to P, if the following condition hold:*

- $\forall a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m, b_1{}', b_2{}', \ldots, b_m{}'\ (P(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m) \wedge P(a_1, a_2, \ldots, a_n, b_1{}', b_2{}', \ldots, b_m{}') \Rightarrow (b_i = b_i{}'))$.

Definition 9. *Consider a semantic segment*

$$S = < S_R, S_A, S_P, S_C.$$

Let Y be the set of all attributes uniquely determined by the attributes in the attribute set X, with respect to S_P. *If* $X \subseteq S_A$, *we say S is an* **extensible semantic segment,** $S_A \cup Y$, *denoted by* $S_A{}^+$ *is called the* **extended attribute set** *of S, and the semantic segment* $S^+ = < S_R, S_A{}^+, S_P, S_C{}^+ >$ *is called the* **extended semantic segment** *of S.*

Since $S_A{}^+$ is uniquely determined by X with respect to S_P, if a tuple consisting of attributes in X satisfies S_P, when extended to contain attributes in $S_A{}^+$, it will also satisfy S_P. This makes it possible to extend S to S^+. Notice that for each extensible segment, there could exist multiple different extended attribute sets, and hence multiple different extended segments. To investigate how to use extended segments in query processing, we examine Example 2 again. Suppose "Ename" is the key of "Employee" relation, thus other attributes of "Employee" can be uniquely determined by "Ename." Hence, S_4 is an extensible segment. Clearly, "Salary" can be uniquely determined by "Ename." To form ${S_4}^+$, we retrieve the tuples containing both "Ename" and "Salary" from the database server and append them to S_{4_C} according to the value of "Ename." After that, Q can be computed from ${S_4}^+$. Therefore, we have Lemma 2.

Lemma 2. *Consider an extensible semantic segment*

$$S =< S_R, S_A, S_P, S_C >,$$

and a query

$$Q =< Q_R, Q_A, Q_P, Q_C >,$$

suppose ${S_A}^+$ is an extended attribute set of S, and S^+ is the extended segment of S with respect to ${S_A}^+$, Q_{P_A} is Q's predicate attribute set. Then, we have:

1 *If $S_R = Q_R$, ${S_A}^+ \cap Q_A \neq \phi$, $Q_P \wedge S_P$ is satisfiable, and $Q_{P_A} \subseteq {S_A}^+$, then Q is answerable by S^+.*
2 *If $S_R = Q_R$, $Q_A \subseteq {S_A}^+$, $Q_P \Rightarrow S_P$, and $Q_{P_A} \subseteq {S_A}^+$, then Q can be fully answered by S^+.*

Proof. Let us construct the extended semantic segment of S, $S^+ =< S_R,\ {S_A}^+, S_P,\ {S_C}^+ >$. According to Lemma 1, consider the relationship between S^+ and Q, we can prove this lemma. □

From Lemma 2, we know that some segments can be used to compute queries after being augmented with the missing attributes. However, it is difficult and time consuming to check whether each cached segment is extensible or not. In what follows, we use more straightforward heuristics to solve computable problems.

Definition 10. *Consider a semantic segment*

$$S =< S_R, S_A, S_P, S_C >,$$

suppose K_A is the primary key of S_R. If $K_A \subseteq S_A$, we say S is a ***key-contained segment****.*

Lemma 3. *Consider a key-contained segment*

$$S =< S_R, S_A, S_P, S_C >,$$

and a query

$$Q =< Q_R, Q_A, Q_P, Q_C >,$$

suppose Q_{PA} is its predicate attribute set. Then, we have:

1. *If $S_R = Q_R$, $Q_P \wedge S_P$ is satisfiable, then Q is answerable by*

$$S^+ =< S_R, S_A \cup Q_A \cup Q_{PA}, S_P, {S^+}_C > .$$

2. *If $S_R = Q_R$, $Q_P \Rightarrow S_P$, then Q can be fully answered by $S^+ =< S_R, S_A \cup Q_A \cup Q_{PA}, S_P, {S^+}_C >$.*

Proof. Let us first prove that S is extensible. Suppose K_A is the primary key of S_R, according to the definition of primary key, we know that all the attributes of S_R can be uniquely determined by K_A. S is key-contained, i.e., $K_A \subseteq S_A$, thus, S is extensible. Further, let ${S_A}^+ = S_A \cup Q_A \cup Q_{PA}$, clearly, ${S_A}^+$ is an extended attribute set of S, and

$$S^+ =< S_R, S_A \cup Q_A \cup Q_{PA}, S_P, {S^+}_C >$$

is the extended segment of S with respect to ${S_A}^+$. According to Lemma 2, consider the relationship between Q and S^+, we can draw the above conclusions.□

4.2 Query Trimming

We consider the query processing problem from two aspects. At first, we look at how to process a query via a single segment, which will be discussed in this section. Then, we examine how to process queries at a cache level, which is the content of the next section. We use Lemma 3 to process queries due to its simplicity and efficiency. To facilitate query processing, we relax Definitions 4 and 5 a little bit to allow semantic segments overlap with each other on key attributes.

Definition 11. *Consider a key-contained segment*

$$S_i =< S_{i_R}, S_{i_A}, S_{i_P}, S_{i_C} >$$

with K_{i_A} being the primary key of S_{i_A}, and a key-contained segment $S_j =< S_{j_R}, S_{j_A}, S_{j_P}, S_{j_C} >$ with K_{j_A} being the primary key of S_{j_A}, we say that S_i and S_j are ***key-contained disjointed*** *if and only if*

1. *$S_{i_A} \cap S_{j_A} \subseteq K_{i_A} \cup K_{j_A}$ or*
2. *$S_{i_P} \wedge S_{j_P}$ is unsatisfiable.*

Definition 12. *A* ***Key-Contained Semantic Cache,*** *C, is defined as $C = \{key-contained\ semantic\ segment\ S_i\}$, where $\forall j, k\ (S_j \in C\ \wedge\ S_k \in C\ \wedge j \neq k \Rightarrow\ S_j$ and S_k are key-contained disjointed).*

When a query Q can only be partially answered by a semantic segment S, we divide it into two parts: one part that can be obtained from S, and the other part which can not be found in S. This procedure is called *Query Trimming*.

Definition 13. *Given a query $Q =< Q_R, Q_A, Q_P, Q_C >$ and a semantic segment $S =< S_R, S_A, S_P, S_C >$,* ***Query Trimming*** *is the process of dividing Q into two subqueries:*

- **Probe Query,** *which retrieves the portion of Q contained in S.*
- **Remainder Query,** *which retrieves the portion of Q not found in S.*

Given a key-contained semantic cache, Fig. 1 illustrates the possible relationships between a query Q and a cached segment S. There exist more scenarios where S is partially or totally contained in Q, however, since these scenarios are symmetric to the cases listed in Fig. 1, the query trimming strategies are very similar.

Case 1: Totally Contained. In this case, $Q_P \Rightarrow S_P, Q_A \subseteq S_A$. Since S is key-contained, according to Lemma 3, we know that Q can be fully answered by S^+. To get S^+, we must retrieve the missing attributes from the server, and we call this query **Amending Query**. However, practically, we will not construct a real S^+. Instead, Q is processed in the following way. If $Q_{PA} \subseteq S_A$, Q can be computed just by applying Q_P to S_C. Otherwise, we use an amending query

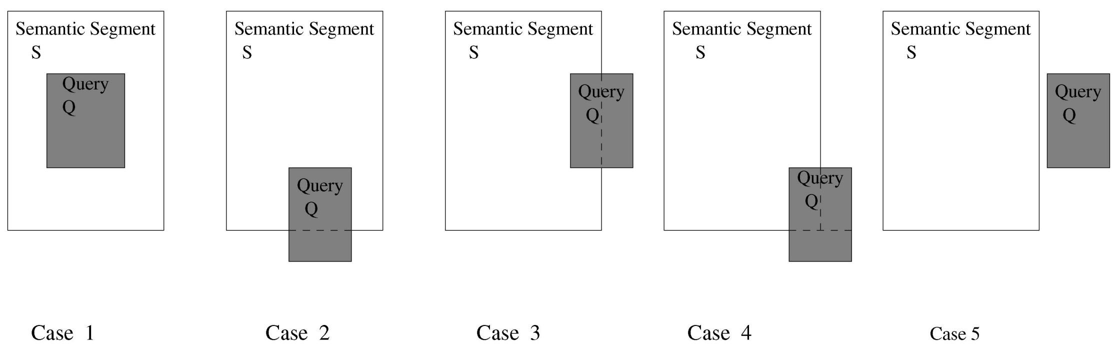


Fig. 1. The possible relationships between Q and S.

AQ = $\pi_{K_A}\sigma_{Q_P}(Q_R)$ to fetch the corresponding values of K_A from the server, then retrieve Q's result by selecting tuples from S according to them.

Case 2: Horizontally Partitioned. Here $Q_A \subseteq S_A, Q_P \wedge S_P$ is satisfiable. Because S is key-contained, according to Lemma 3, we know that Q is answerable by S^+. Since a subset of Q's tuples is contained in S, we trim Q horizontally, i.e., trim Q's predicate into two parts:

- $Q_1 = Q_P \wedge S_P$, the subset of Q contained in S;
- $Q_2 = Q_P \wedge \neg S_P$, the subset of Q not contained in S.

As Q's result will be admitted into the cache, to make the new semantic segment key-contained, besides Q's original attributes, the new segment should also include K_A. Hence, both the probe and remainder query will retrieve tuples with attribute set $Q_A \cup K_A$. Also, because K_A helps to select resulting tuples, we won't actually construct S^+. We use the same amending query as Case 1.

Case 3: Vertically Partitioned. In this scenario $Q_P \Rightarrow S_P$, but $Q_A \subseteq S_A$ does not hold. Because S is key-contained, according to Lemma 3, we know that Q can be fully answered by S^+. As before, we do not build S^+ here. Since only a subset of Q's attributes is contained in S, we trim Q vertically, i.e., trim Q's attributes into two parts:

- $A_1 = (Q_A \cap S_A) \cup K_A$, the subset of Q's attributes contained in S,
- $A_2 = (Q_A - S_A) \cup K_A$, the subset of Q's attributes not contained in S.

The probe query could simply be $\pi_{A_1}(S_C)$, since, after being appended to the remainder query result according to K_A, the part of its result that does not satisfy Q will be removed.

Case 4: Hybridly Partitioned. Here, $Q_P \wedge S_P$ is satisfiable, but $Q_A \subseteq S_A$ does not hold. Because S is key-contained, from Lemma 3, we know that Q is answerable by S^+. We trim S hybridly, i.e., do a horizontal trimming on Q first, and then do a vertical trimming on the resulting probe query. Following the trimming mechanism for the horizontally partitioned case, we have:

- Probe Query: $PQ_1 = \pi_{Q_A \cup K_A}\sigma_{Q_P}(S_C)$, processed on S.
- Remainder Query: $RQ_1 = \pi_{Q_A \cup K_A}\sigma_{Q_P \wedge \neg S_P}(Q_R)$, processed at the server or further trimmed by other segments.

Since $Q_A \subseteq S_A$ does not hold, PQ_1 could not be directly computed from S. Hence, we do a vertical trimming on PQ_1, using the strategy discussed in Case 3.

Case 5: Not Contained. Here, $Q_P \wedge S_P$ is unsatisfiable, i.e., S does not contain any result of Q. Since Q cannot be computed from S, the probe query is NULL, and the remainder query is Q, itself. However, because Q's result will be cached, to make the resulting segment key-contained, the key attribute set K_A should be included in the final result.

To summarize the query trimming work discussed in this section, we present Algorithm 1, which trims a query Q via a key-contained semantic segment S which is defined on the same relation as Q.

Algorithm 1 QueryTrim (Query Q, Segment S, Query pq, Query aq, Query rq1, Query rq2, int type), to trim a query Q via a semantic segment S.

Input: Query Q; key-contained semantic segment S

Output: Probe Query pq; Amending Query aq; Remainder Query rq1, rq2; Trimming Type type

Procedure: {

```
    $K_A \leftarrow$ S's key attribute set;
    $A_1 \leftarrow (Q_A \cap S_A) \cup K_A; A_2 \leftarrow (Q_A - S_A) \cup K_A$;
    IF ($Q_A \subseteq S_A$) {
        IF ($Q_P \Rightarrow S_P$) {
            /***** Case 1: Totally Contained *****/
            type = 1;
            IF ($Q_{PA} \subseteq S_A$) THEN aq = NULL; ELSE
            aq = $\pi_{K_A}\sigma_{Q_P}(Q_R)$;
            pq = $\pi_{Q_A \cup K_A}\sigma_{Q_P}(S_C)$
            rq1 = rq2 = NULL; return; }
        IF ($Q_P \wedge S_P$ is satisfiable) {
          /***** Case 2: Horizontally Partitioned *****/
          type = 2;
          IF ($Q_{PA} \subseteq S_A$) THEN aq = NULL; ELSE
          aq = $\pi_{K_A}\sigma_{Q_P \wedge S_P}(Q_R)$;
          pq = $\pi_{Q_A \cup K_A}\sigma_{Q_P}(S_C)$;
          rq1 = $\pi_{Q_A \cup K_A}\sigma_{Q_P \wedge \neg S_P}(Q_R)$;
          rq2 = NULL; return; } }
    IF ($Q_A \subseteq S_A$) does not hold {
        IF ($Q_P \Rightarrow S_P$) {
          /***** Case 3: Vertically Partitioned *****/
          type = 3; pq = $\pi_{A_1}(S_C)$; rq1 = $\pi_{A_2}\sigma_{Q_P}(Q_R)$;
          rq2 = aq = NULL; return; }
```

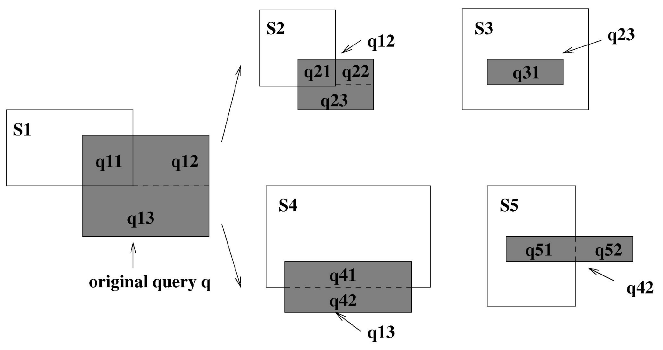


Fig. 2. Example 3: A semantic cache query processing example.

```
        IF ($Q_P \wedge S_P$ is satisfiable) {
            /***** Case 4: Hybridly Partitioned *****/
            type = 4;
            pq = $\pi_{A_1}(S_C)$; rq1 = $\pi_{Q_A \cup K_A} \sigma_{Q_P \wedge \neg S_P}(Q_R)$;
            rq2 = $\pi_{A_2} \sigma_{Q_P \wedge S_P}(Q_R)$; aq = NULL; return; }  }
    /***** Case 5: Not Contained *****/
    rq1 = Q; \ pq = aq = rq2 = NULL; type = 5; return; }
```

4.3 Query Processing

While every individual segment may contain only a small part of a query result, multiple segments can be combined together to generate a much bigger part of, or even the whole, result. In this section, we extend the query processing mechanism to the case when the entire cache is considered. After a query is trimmed by the first segment, the remainder query is then trimmed by the next candidate segment. This process continues until there is no segment that could contribute to the query result. And, the final remainder query will be sent to the server for processing. We illustrate this procedure with Example 3.

Example 3. In this example, the semantic caching query processing procedure is shown in Fig. 2. At first, the query q is hybridly trimmed by segment S_1, thus, we have one probe query q11 processed on S_1, and two remainder queries, q12 and q13. Assume that q12 is then, hybridly trimmed by segment S_2, we have another probe query q21 processed on S_2, and two remainder queries, q22 and q23. Meanwhile, q13 is horizontally trimmed by S_4, resulting in the probe query q41 and the remainder query q42. Moreover, q23 and q42 are further trimmed by S_3 and S_5, respectively. Suppose, at this moment, no more segments can be found to answer the remainder queries. Thus, q22 and q52 have to be sent to the server side to be processed.

From this example, we can see that the query processing procedure gets more difficult when the remainder queries are allowed to be trimmed by the semantic segments. Things become extremely complicated when the hybridly partitioned case occurs, since, in such a scenario, two remainder queries are generated. Therefore, we propose a data structure, *query plan tree*, which clearly expresses the relationship between every part of the query and the involved semantic segments. It is implemented as a binary tree. Each node of a query plan tree is described using five

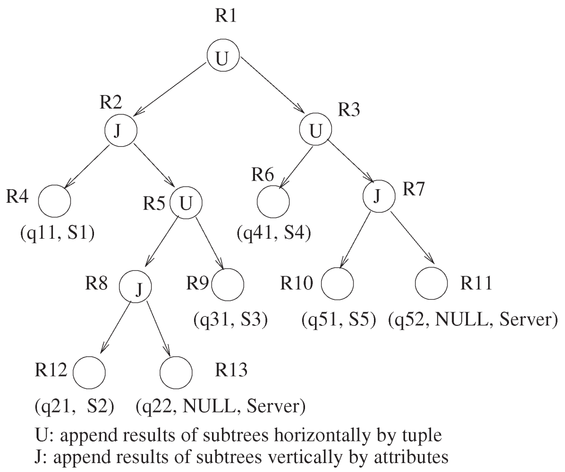
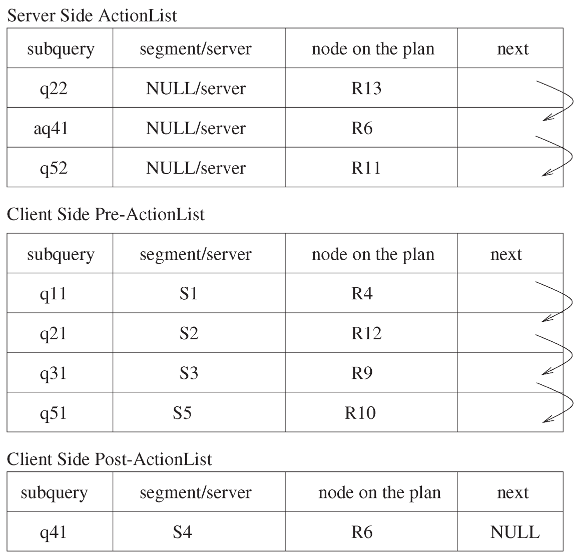


Fig. 3. Example 4: A query plan tree.

attributes. The attribute "type" indicates whether that node is a leaf node, a "union" node or a "join" node. "Subquery" gives the subquery related with the node, and "seg" shows the associated semantic segment if "subquery" is a probe query. If "seg" is NULL, that means "subquery" should be processed at the server side. Notice that the functionality of internal nodes (nodes with types of "union" or "join") is to memorize the query computing path, they are not used to stand for the actual query processing operations. Therefore, the attributes "subquery" and "seg" of all internal nodes are not recorded and set to NULL. At last, "left" and "right" define the left and right children, respectively. A query plan tree is created simultaneously with the query trimming procedure, which we will illustrate using Example 4.

Example 4. In this example, a query plan tree, which is constructed simultaneously with the query processing procedure described in Example 3, is shown in Fig. 3. Consider the case when q is hybridly trimmed by S_1. After q is horizontally trimmed, node R_1 is created with the key "U," which means that the results of the subtrees must be appended horizontally by tuple. Also, R_1's subtrees represent the two trimmed parts, respectively. Then, the left subtree is vertically trimmed, thus node R_2 is created. Here, the key of R_2 is "J," which indicates that the results of R_2's subtrees are appended vertically by attributes. Until now, the left subtree of R_2 cannot be further trimmed, it stands for the probe query q11 processed on S_1. Similarly, the query plan tree grows up as the query is further trimmed by S_2, S_3, S_4, and S_5.

With the complete query plan tree, we can begin to compute the query. There exists several different ways to process a query. One straightforward approach, called *sequential query processing*, is to process each subquery in sequence. Obviously, this is not efficient, especially in the cases when the amending query results from the server are needed. On the other hand, since we can immediately determine the exact set of subqueries to be processed on the server after reasoning among the query description and the semantic information about the cached segments maintained in the cache index, we could let the client and server work in parallel. We call this *parallel query processing*. To do



Server Side ActionList

subquery	segment/server	node on the plan	next
q22	NULL/server	R13	
aq41	NULL/server	R6	
q52	NULL/server	R11	

Client Side Pre-ActionList

subquery	segment/server	node on the plan	next
q11	S1	R4	
q21	S2	R12	
q31	S3	R9	
q51	S5	R10	

Client Side Post-ActionList

subquery	segment/server	node on the plan	next
q41	S4	R6	NULL

Fig. 4. Example 5: An ActionList example.

this, we need an additional auxiliary data structure, *ActionList*, to keep track of the queries which are processed on the server and the client sides, respectively. An ActionList is a list of Actions. Each Action is described by four attributes. "Subquery" gives the query that needs to be processed. "seg" indicates the related semantic segment when "subquery" is a probe query. If "subquery" must be processed at the server side, "seg" is NULL. "Node" associates this action with the specific node on the query plan tree. "Next" links this action to the next action in the same ActionList. We further explain the concept of ActionList in Example 5.

Example 5. In this example, the action lists that are generated according to the query plan tree, described in Example 4, are illustrated in Fig. 4. The server side action list includes all the subqueries that must be processed on the server side. The client side preaction list contains the subqueries that can be computed without the help of the server, while the postaction list shows those subqueries that must wait for the results from the server to be processed. Suppose, in Example 3, segment S_4 does not contain the predicate attributes of q13. Thus, an amending query aq41 is needed in order to answer q41 from S_4. We insert aq41 in the server action list, and put q41 in the postaction list. Obviously, the server can process the subqueries in the server action list at the same time when the client works on the preaction list. By doing so, the system performance is expected to be improved.

As we have indicated before, in a key-contained semantic cache, the segments are disjointed with each other except the key attributes. Thus, for a specific query q, the set of segments that contribute to the result of q is unique, namely, the nodes in q's plan tree can be uniquely determined. In what follows, we give the recursive algorithm for generating a query plan and the associated action lists. For a query Q, to obtain the query plan tree *plan*, the server action list srv_act, and the related client side action list pre_clt and $post\_clt$, we call Gen_Plan(Q, NULL, NULL_Direction, srv_act, pre_clt, post_clt, plan).

Algorithm 2 Gen_Plan(Query q, QueryPlan parent, int direction, ActionList srv_act, ActionList pre_clt, Action List post_clt, QueryPlan plan), to generate a query plan and related action lists for query q from the semantic cache C

Input: q: the query; parent: the parent node in the plan tree; direction: which kid (left or right) the current node will be.

Output: srv_act: the sever side action list; pre_clt: the client side preaction list; post_clt: the client side action list; plan: the final query plan.

Procedure: Create a new node "kid." If parent = NULL, we have plan = kid. Otherwise, if direction = "left," we let "kid" be the left child of "parent," and if direction = "right," we let "kid" be the right child of "parent." Find a segment S in the semantic cache which hasn't been visited and can answer q, then check the following cases.

- If there exists such an S, mark S as "used." Call QueryTrim(q, S, pq, aq, rq1, rq2, type). Then, we deal with the following scenarios.
  - If q is totally contained in S, i.e., type = 1, we associate "kid" with q and S. If aq $\neq$ NULL, we insert aq in srv_act, and put the corresponding probe query pq and S in post_clt. Otherwise, we put pq and S in pre_clt.
  - If q is horizontally partitioned by S, i.e. type = 2, we associate "kid" with "U". Create a new node "leftgrandkid," associate it with the probe query pq and S. Let "leftgrandkid" be the left kid of "kid." If aq $\neq$ NULL, we insert aq in srv_act, and put pq and S in post_clt. Otherwise, we put them in pre_clt. Then for the remainder query rq1, recursively call this procedure Gen_Plan (rq1, kid, "right," srv_lst, pre_lst, post_lst, plan).
  - If q is vertically partitioned by S, i.e., type = 3, we associate "kid" with "J." Create a new node "leftgrandkid," associate it with the probe query pq and S. Let "leftgrandkid" be the left kid of "kid." If S contains the predicate attributes of q, insert pq and S in pre_lst; otherwise, insert them in post_lst. Then for the remainder query rq1, recursively call this procedure Gen_Plan(rq1, kid, "right," srv_lst, pre_lst, post_lst, plan).
  - If q is hybridly partitioned by S, i.e., type = 4, we associate "kid" with "U." Create a node "leftgrandkid," associate it with "J," and let it be the left child of "kid." Further create another node "leftgrandgrandkid," associate it with the probe query pq and S, and let it be the left child of "leftgrandkid." If S contains the predicate attributes of q, insert pq and S in pre_lst; otherwise, insert them in post_lst. Then, for the two remainder query rq1 and rq2, call first Gen_Plan(rq2, leftgrandkid, "right," srv_lst,

pre_lst, post_lst, plan), then call Gen_Plan(rq1, kid, "right," srv_lst, pre_lst, post_lst, plan).

- If there does not exist such an S, associate node "kid" with q, and indicate that it must be processed at the server side, and add the corresponding item in srv_act.

When all the subquery results are obtained, the query plan tree and the action lists are used to combine all partial results together to generate the final result. From Algorithm 2, we notice that the root nodes are always generated before their subtrees, namely, the query plan tree is built in preorder. However, the result coalescing procedure takes a reverse order. The final query result is obtained by a postorder traversal of the query plan tree. This makes sense since the result of a parent node can be calculated only when the results of its subtrees are known. Thus, we propose Algorithm 3 to compute the final result using the plan tree. Suppose, that the query plan for a query Q is *plan*, *stack* is a temporary stack, the result of Q *res* can be obtained by calling Com_Res (plan, plan, stack, res).

Algorithm 3 Comp_Res(QueryPlan node, QueryPlan root, ResultStack stack, QueryResult result), to compute query result using query plan tree.

Input: node: the current node being processed, root: the root of the query plan tree; stack: the result stack;

Output: Res: the final query result.

```
Procedure: IF (node != NULL) {
    Comp_Res(node→ left, root, stack, result);
    Comp_Res(node→ right, root, stack, result);
    Switch ( node ) {
        Case a leaf node:
            res ← get the associated query result;
            stack ← push (res); break;
        Case a "U" or "J" node:
            res1 ← pop (stack); res2 ← pop (stack);
            IF (node = "U") THEN res ← append res1 and
            res2 by tuple
            IF (node = "J") THEN res ← append res1 and
            res2 by attribute
            stack ← push (res); break;
        Case a root node:
            result ← pop(stack); break; } }
```

In summary, the semantic cache query processing involves three steps:

1. to generate the query plan,
2. to process each subquery against the segment or the database, and
3. to combine results together by traversing the plan tree.

Thus, to analyze the time complexity in query processing, we need to examine the time complexity of each step. However, it is very difficult to precisely quantify the time complexity since it depends on the database physical organization, relational operator evaluation strategies, the cache size, how complicated the query is, and how it is related to the semantic cache. Hence, in the following, we only briefly state how to analyze the time complexity for each step, without giving the exact results.

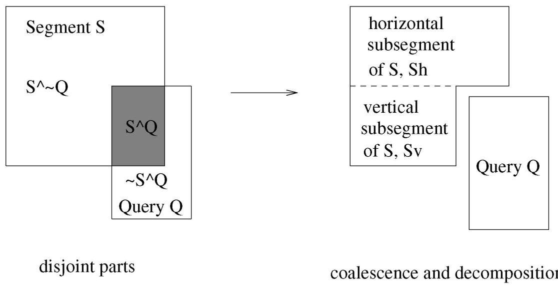

Segment S
S^~Q
S^Q
~S^Q
Query Q
horizontal subsegment of S, Sh
vertical subsegment of S, Sv
Query Q
disjoint parts
coalescence and decomposition

Fig. 5. Handling disjoint parts after query trimming.

Suppose the query plan tree generated by step 1 has *n* nonleaf nodes (the value of n is determined by the semantic relationship between the query and the cache). For each such node, we call 0 or 1 times the QueryTrim procedure. So the time complexity of step 1 is O(n*TmComplex(QueryTrim)), where TmComplex(QueryTrim) stands for the time complexity of query trimming. From Algorithm 1, we can see that the query trimming procedure sequentially check implicity/satisfiability among the query and the cached segments. So, TmComplex(QueryTrim) is determined by the number of cached segments and the time complexity of the implicity/satisfiability checking algorithm which is examined in the references such as [22] and [14].

Obviously, the time complexity of step 2 really depends on the relational operator evaluation strategy and the database physical organization such as indexing and clustering. Step 3 is very straightforward. To make the problem simple, we assume that the "join" and "union" operations take the same time *C* no matter the sizes of join/union parties. Since every nonleaf node of the query plan tree does a join/union operation exactly once, the time complexity of step 3 is approximately O(n).

4.4 Coalescence and Decomposition

In this section, we deal with the scenario after a query trimming. Fig. 5 shows such a typical scenario when a query Q is hybridly partitioned by a semantic segment S. Though there exist other cases, they are similar in principle.

To avoid redundant data, we can not keep both S and Q in the cache. One way to do so is through decomposition and coalescence. As shown in the left part of Fig. 5, three disjoint parts are left after query trimming. Then, we have the following options:

- **Partial Coalescence in Query:** To decompose segment S into two parts: the overlapped part and nonoverlapped part. Then, to coalesce every part of Q together and cache the result of Q.
- **Partial Coalescence in Segment:** To keep the original segment S, and to decompose Q into two parts. Only the part of Q which is not in the cache will be admitted.
- **No Coalescence:** To leave three disjoint parts separate.

The "no coalescence" approach reflects the frequency of reference at a finer granularity. But the disadvantage is that it may result in a large number of small semantic segments, which increases the overhead of cache management and query processing. The "partial coalescence in segment"

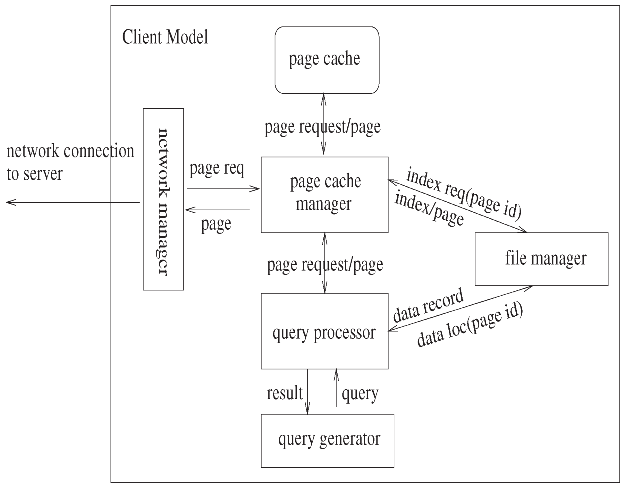


Fig. 6. The page caching client.

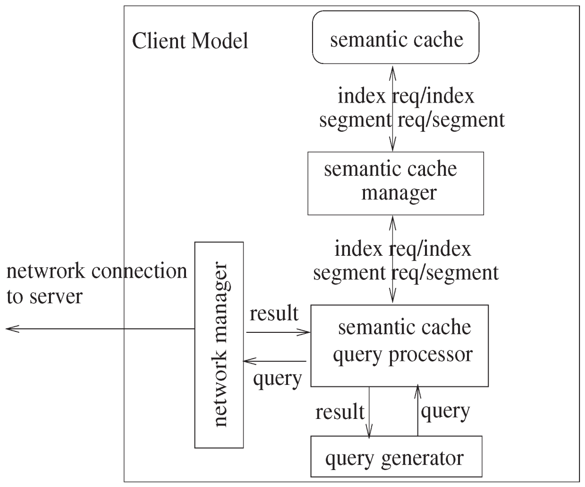

Client Model
semantic cache
index req/index segment req/segment
semantic cache manager
index req/index segment req/segment
netwrork connection to server
network manager
result
query
semantic cache query processor
result
query
query generator

Fig. 7. The semantic caching client.

approach can not effectively express the frequency of reference. The old part of the segment has a small chance to be replaced even though it may never be used. The "partial coalescence in query" approach seems to be the best among all these choices, since it avoids the disadvantages of others. In this research, we use this strategy to do decomposition and coalescence.

After the overlapped part is removed from S, the part left is irregular. It is very difficult to use a single semantic description to describe it. Hence, we divide it into two parts: the *horizontal subsegment* S_h and the *vertical subsegment* S_v. The definitions of these two segments can be generated during the query trimming procedure. Given a semantic segment $S =< S_R,\ S_A,\ S_P,\ S_C >$, and a query $Q =< Q_R, Q_A, Q_P, Q_C >$, assume that Q can be trimmed by S, we derive the definitions of S_h and S_v in the following.

- **horizontal subsegment**: $S_h = < S_R,\ S_A,\ S_P \wedge\ \neg Q_P,\ S_{hC} >$.
- **vertical subsegment:** $S_v = < S_R,\ (S_A - Q_A) \cup K_A,\ S_P \wedge\ Q_P,\ S_{vC} >$.

5 Performance Study

We examine the performance of the semantic caching model through a simulation study. As page caching is one of the most widely used caching schemes, we choose it as the technique to compare with. Our simulator is implemented in C++ using CSIM ([4]), which simulates a simple but typical client-server model. In this section, we first discuss the simulation environment, then present the conducted experiments and analyze the results.

5.1 Simulation Model

In this study, the system is modeled to be composed of a server, a client and a network connecting them. This model can be used to simulate a mobile system when a small network bandwidth value is specified. The server maintains a complete copy of the database and acts as a database server, while all the queries are submitted by the client. To compare the performance of different caching schemes, we simulate either a page cache or a semantic cache on the client side. In the following, we give the server and the client model, respectively. The system parameters and the workload design are specified as well.

5.1.1 Client Model

Two types of client models are simulated: the page caching client (Fig. 6) and the semantic caching client (Fig. 7). Both clients work under the same query workload which is generated by the *Query Generator* module. They also use the same mechanism to manage communications with the server implemented by the *Network Manager* module. But the two clients differ in query processing and cache management.

In page caching, queries are processed at the client side with data shipped from the server. If there is an index on the attribute queried, the client first accesses the index and then accesses the qualifying data pages. Otherwise, it has to scan the whole database. The *File Manager* module is responsible for determining the physical locations for requested data items. To simplify the problem, the index is simulated in the way that only the space and runtime overheads of the leaf level are considered. The *Cache Manager* module manages the cache at a granularity of page. If a requested page is not in the cache, it will be fetched from the server via the network. The space overheads for page cache management, such as Buffer Control Blocks ([12]), are simulated. At last, queries are processed by the *Query Processor* module.

For a semantic caching client, the cache is managed by the *Semantic Cache Manager*. To maintain fairness, the space overheads for cache management, such as index, are counted. Since the management information for semantic caching is maintained at a granularity of semantic segments, which keeps on changing as time passes by, the space overhead is variable. Our approach simply makes sure that the size of management overhead and actual data doesn't exceed the cache size. The *Semantic Cache Query Processor* module processes queries via the cache, the parallel as well as sequential query processing strategies are simulated.

5.1.2 Server Model

The server receives messages from the client and processes them. A page caching client sends a page request to the server, who fetches the page and sends it back. Things get

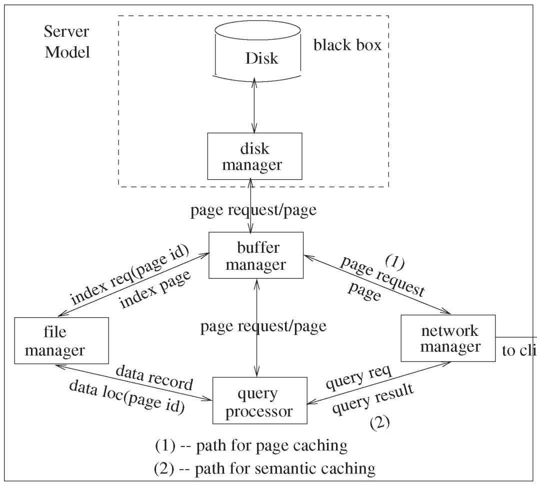


Fig. 8. The server model.

complicated for a semantic caching client. The client can send a single query (sequential query processing) or a set of queries (parallel query processing) to the server. The server processes the query and sends the result back.

To be fair with different scenarios, we assume that, initially, the server side buffer does not contain any data pages needed for answering a query. Also, the server follows the same access path (file scan or through index) as the page caching client. In the parallel processing case, a set of subqueries will be sent to the server altogether. If the access path is a file scan, these queries will share one file scan. This is reasonable, since we can always assume that there exists a query optimizer on the server side. If there is an index, the queries will be processed one by one. However, the subsequent subqueries, except the first one, can make use of the server side buffer which uses LRU as the replacement policy. In the sequential processing case, every time the client sends only one subquery to the server. The server processes the query in a similar way as a page caching client.

In summary, the server is composed of the following modules (Fig. 8). The *Query Processor* processes queries from the client. The *Buffer Manager* manages the memory buffer at a granularity of page. A requested page which is not in the buffer is fetched from the disk. The *File Manager* determines the physical locations for required data items. The *Disk Manager* is responsible for accessing data pages from the disk. At last, the *Network Manager* manages the communications with the client.

5.1.3 System and Workload Parameters

The first part of Table 3 illustrates the primary system parameters used in the simulation study. Both the server and the client CPUs adopt a FCFS scheduling policy, their speeds are described by *ServerMips* and *ClientMips*, respectively. *ServerCache* and *ClientCache* define the sizes of the memory caches at the server and the client. The network bandwidth is given by *BandWidth*. When *Bandwidth* is small, a mobile system is simulated. In this study, we assume that mobile clients are relatively powerful computers such as laptops.

TABLE 3
System and Workload Parameter Settings

Parameter	Description	Value
ServerMips	server CPU speed (Mips)	50
ServerCache	server side cache size (kb)	256
ClientMips	client CPU speed (Mips)	50
ClientCache	client side cache size (kb)	128
BandWidth	network bandwidth (Mb/s)	8/1/0.5
PageSize	size of data page (bytes)	4096
FixMsgIns	fixed part protocol cost to send/receive a message	20000
PerByteMsg	size-dependent part protocol cost to send/receive message	3
StartIO	number of CPU instructions to start an IO operation	5000
Compare	number of CPU instructions to compare	2
CopyWord	number of CPU instructions to copy a word	1
HashCost	number of CPU instructions to do a hash function	9
DiskTm	time to read a page from disk (ms)	12
RelSize	size of a relation (tuples)	10,000
TupleSize	size of a tuple (bytes)	200
AttrSize	size of an attribute (bytes)	4
TupleHot	the percentage of hot tuples	0.2
AttrHot	the percentage of hot attributes	0.2
SelRag	the selectivity of a query	100 - 1,000
NumProjAtt	the projectivity of a query	10 - 50
SelAtt	the attribute that is queried	A1, A2, A3
ProjectIn	whether the queried attribute is projected	T/F
Skew	the percentage of queries that access hot tuples/attributes	0.1 - 1
WarmUpQry	number of warm-up queries	50
TotalQry	number of queries reported in result	500

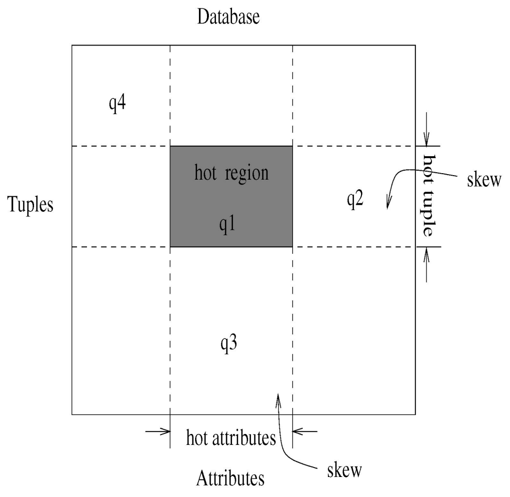


Fig. 9. The query workload.

Database relations, cached pages, semantic segments, and cache maintenance data are all physically stored in pages, whose size is specified as *PageSize*. The cost model used in the simulation is mainly taken from [1]. The network is modeled as a FIFO queue. The cost of sending/receiving a message involves the time-on-wire (transfer time), and the time for protocol including a fixed part (*FixMsgIns*) and a size-dependent part (*PerByteMsg*). *StartIO* gives the CPU cost for starting a disk I/O operation, and *DiskTm* specifies how long it takes to read a page from the disk. Only the costs of basic operators are listed in Table 3, the cost for more complicated operations such as predicate computation can be easily derived. *HashCost* gives the CPU cost to do a hash function.

We use a modified Wisconsin Benchmark ([13]) to examine the performance. The benchmark database contains one single relation, R, with 10,000 tuples, each tuple has a size of 200 bytes. For ease of control the projectivities of queries, we let R consist of 50 4-byte integers instead of the original benchmark scheme with 13 4-byte integers and three 52-byte character strings. To study the performance under different database physical organizations, we assume that the relation has three key attributes: A1 is indexed but unclustered, A2 is both indexed and clustered, and A3 is neither indexed nor clustered. The database is divided into two regions: the *hot region* and the *cold region*. The size of the hot region is defined by *TupleHot* and *AttrHot*, the remainder part of the database is regarded as the cold region.

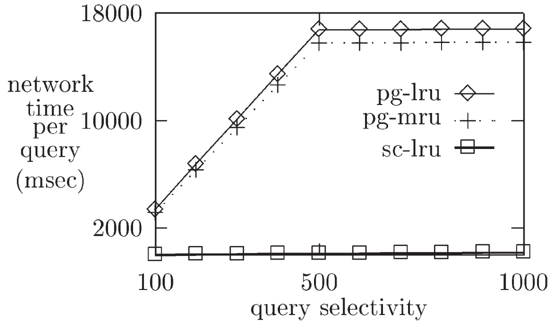


Fig. 11. SelAtt = A1, ProjIn = T, Skew = 0.8.

Queries used in this study involve only select and project operations. For ease of implementation, we assume that each select condition is a disjunction of conjunctions of compare predicates which are defined on one single attribute. The size of a query is specified by *SelRag* and *NumProjAtt* which are the selectivity and projectivity of the query, respectively. Moreover, we use *SelAtt* to select which attribute to be queried, and *ProjectIn* to determine whether to always project that attribute. By doing so, different scenarios that favor either page or semantic caching can be generated.

Queries are randomly generated. For each newly created query, its select predicate is first determined according to the rule that it has a probability of *Skew* to have the semantic centerpoint within the hot region. Then, we choose its projected attributes which also have a probability of *Skew* to be in the hot database attributes. Fig. 9 further illustrates such a point: q1 accesses both hot tuples and attributes, q2 visits hot tuples but cold attributes, q3 accesses cold tuples but hot attributes, and q4 visits both cold tuples and attributes. Notice that, when the number of projected attributes is bigger than the number of hot attributes, all the queries will involve at least part of the cold attributes. For each run of simulation, we use *WarmUpQry* queries to warm up the cache, and process *TotalQry* queries to actually collect the performance metrics.

5.2 Experiments

The experiments conducted are designed to compare the performance of page caching and semantic caching. We use response time as the primary performance metric, while

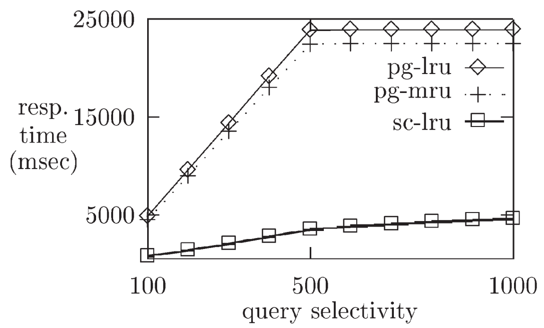


Fig 10. SelAtt = A1, ProjIn = T, Skew = 0.8.

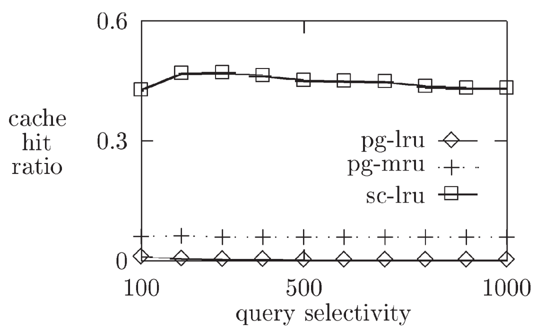


Fig. 12. SelAtt = A1, ProjIn = T, Skew = 0.8.

TABLE 4
Semanti Cache when Select on A1, NumProjAtt = 10, ProjIn = True, Skew = 0.8, BandWidth = 1Mb/s

selectivity	num of cached segments	time consumed at client	avg response time
100	30	23	736
200	17.6	44	1358
300	12.1	65	2029
400	9.2	85	2746
500	7.3	104	3479
600	6.1	125	3773
700	5.2	142	3993
800	4.4	159	4232
900	3.9	179	4440
1000	3.5	199	4560

other parameters such as network traffic and cache hit ratio are also examined. The presented results are obtained by averaging the results of three runs of simulation. The only difference between each run is that different random numbers are used to control the query workload. Since these random numbers are generated following the same distribution, there is not much deviation between the simulation results across three runs.

5.2.1 Impact of Database Physical Organization

In this part, we examine how the cache performance is impacted by database physical organization: indexing and clustering. Fig. 10, Fig. 11, and Fig. 12 compare page with semantic caching when A1, which is an indexed but unclustered attribute, is queried. The parameters are set the same as shown in Table 3, except that *ProjectIn* is "True," *Skew* is 0.8, *NumProjAtt* is 10, and *BandWidth* to 1Mb/s.

The x-axis of these figures represent the query selectivity, which ranges from 100 to 1,000. When the value of *NumProjAtt* is fixed, the x-axis actually stands for the query size. The y-axis of Fig. 10 shows the average query response time, which is defined as the interval between the time when a query is issued and the time when its result is obtained. Fig. 11 gives how the network traffic changes as the query size increases. The y-axis represents the average network time consumed per query. Fig. 12 shows the average cache hit ratio. For page caching, it is defined as the ratio of the number of pages that are found in the cache to the number of all pages that a query has accessed. While for semantic caching, it is the percentage of the result of a query that is located in the cache.

Two replacement strategies, LRU and MRU, are used in page caching, while LRU is used in semantic caching. Fig. 10 shows that the response time for all three parties worsens as the query size increases. This is because more data has to be fetched from the server disk and transferred over the network, which is also observed from Fig. 11. Since A1 is indexed but unclustered, a larger selectivity query usually involves more qualified pages. Hence, a page caching client needs to bring more data pages from the server. An interesting notice is that both the response time and network time increase very slowly after the query selectivity is larger than 500. This can be explained by the structure of the benchmark database R. As we know, R has 10,000 tuples, each tuple is 200 bytes long, hence, R can be stored in 500 pages if the page size is set to 4,096. For a totally unclustered attribute A1, when the selectivity is bigger than 500, the number of qualified pages accessed is always the same: 500. On the other hand, a semantic cache client sends the remainder queries to the server, who processes queries by fetching data pages from the disk. After the query selectivity reaches a certain point, the server has to fetch all data pages to answer the query. This explains why the semantic caching curve in Fig. 10 also becomes more stable after the query selectivity is larger than a certain value.

We notice that semantic caching completely outperforms page caching. This is due to two reasons. First, the semantic cache hit rate is much higher than that of page caching (Fig. 12). For page caching, to process a query, all qualified pages must be visited even though each of them may contain only a small part of the query result. Second, the network traffic of semantic caching is much lower than that of page caching (Fig. 11). A lot of unuseful data are sent from the server in page caching.

MRU has a relatively better performance than LRU in page caching. When the skew rate is high, say 0.8, the consecutive queries may overlap in their predicates. Hence, they might share some index and data pages. As index pages are always accessed first, when the cache size is small, MRU is more effective than LRU to keep these useful pages in the cache.

The performance differences between these schemes increase as the network bandwidth decreases. For example, when *BandWidth* is 19.2kbps, which is a typical wireless link bandwidth, for queries of selectivity 100, the average response time for page LRU is 176,188, page MRU is 166,941, and that of semantic cache is only 1,743.

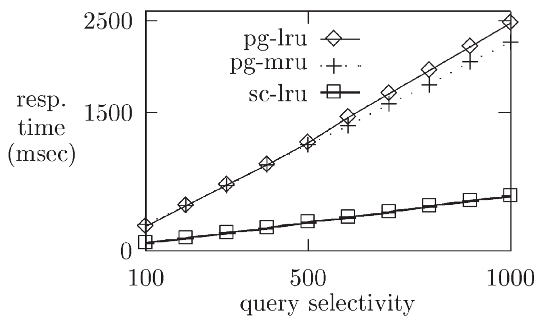


Fig. 13. SelAtt = A2, ProjIn = T, Skew = 0.8.

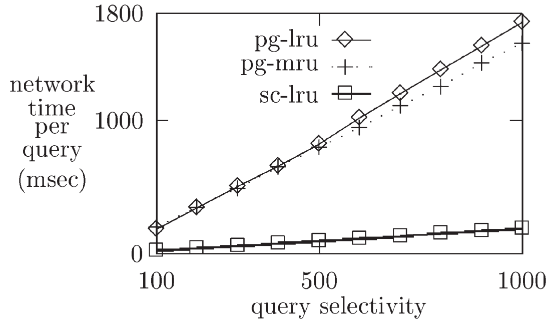


Fig. 14. SelAtt = A2, ProjIn = T, Skew = 0.8.

We can see that semantic caching shows advantages in mobile computing.

Some additional semantic caching performance results are given in Table 4. We can see that the cost at the client side is very low compared to the total cost. This is due to two reasons. First, all the operations at the client side are done in memory. Second, the queries examined in this study are relatively simple, they only involve select and project operations. Also, there are not too many segments in the cache. Actually, the number of segments in the cache is impacted by the query size. The replacement strategy ensures that the segments less recently used to be always replaced.

Fig. 13, Fig. 14, and Fig. 15 illustrate the scenario when the selection is on attribute A2, which is indexed and fully clustered. Fig. 13 shows that the response time of all three parties becomes worse as the query selectivity increases. Because A2 is indexed, the number of qualified pages that a query needs to visit increases linearly as the selectivity increases. Page caching still performs worse than semantic caching in this experiment. The difference between the two schemes gets bigger as the query selectivity increases. For page caching, the data granularity of client-server transfer is a data page. When the projection is involved, even if a fully clustered attribute is selected, the client still has to fetch a lot of unused attributes from the server, which causes lots of network traffic (Fig. 14). Semantic caching is more effective in utilizing cache space due to projection. Hence, its hit rate is higher than that of a page cache (Fig. 15). As the performance of semantic caching is mainly impacted by the nature of query workload, when *Skew* is fixed to 0.8, its hit rate is relatively constant (Fig. 15). The hit rate of page caching decreases as the selectivity increases if LRU is used. This is reasonable, since the cache size remains unchanged, and more pages need to be visited. When MRU is used, the hit rate also decreases, but at a very slow speed. This can be explained the same way as we explain Fig. 12. Overall, all three approaches behave much better in this experiment than in the previous one when A1 is queried.

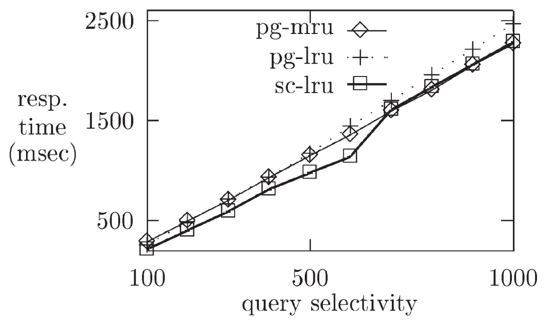


Fig. 16. SelAtt = A2, NumProjAtt = 50, Skew = 0.8.

To further demonstrate the impact of projection, we study the scenario when *NumProjAtt* is set to 50, i.e., all attributes are projected. Fig. 16 illustrates the performance results in terms of response time. We notice that the performance of semantic caching is similar to that of page caching. If all attributes are projected, the advantages of semantic caching due to projections are lost. When *SelRag* is less than 700, semantic caching still performs slightly better than page caching. Since a page cache is managed at a page level, even when A2 is fully clustered, there are still some extra tuples in a page sent over the network and stored at the cache. In contrast, a semantic cache only requests and keeps the exact data items needed. When *SelRag* equals to or is bigger than 700, page caching outperforms semantic caching. This is because when *SelRag* is 700, the query size is larger than the cache size. Since a query result can only be admitted to a semantic cache as a whole, this result will not be admitted. Hence, no segments are cached in the semantic cache after this point. While subsequent queries can still visit some pages from the local page cache, they must be totally processed at the server side in a semantic caching scenario. From this experiment, we notice that the performance of semantic caching will be impacted if the cache granularity is too coarse.

Fig. 17, Fig. 18, and Fig. 19 show the results when A3, an unindexed and unclustered attribute, is selected. For page caching, since A3 is unindexed, the query access path is a

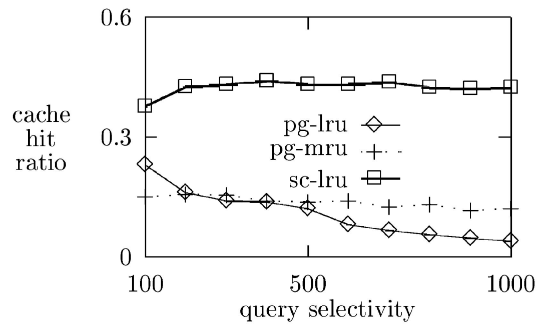


Fig. 15. SelAtt = A2, ProjIn = T, Skew = 0.8.

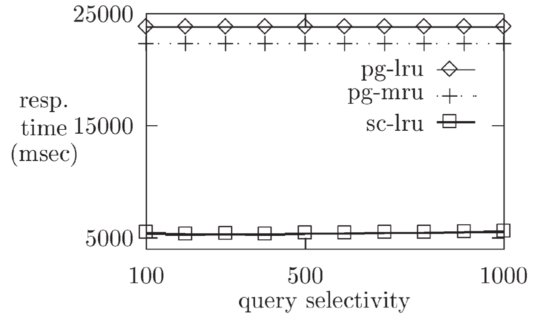


Fig. 17. SelAtt = A3, ProjIn = T, Skew = 0.8.

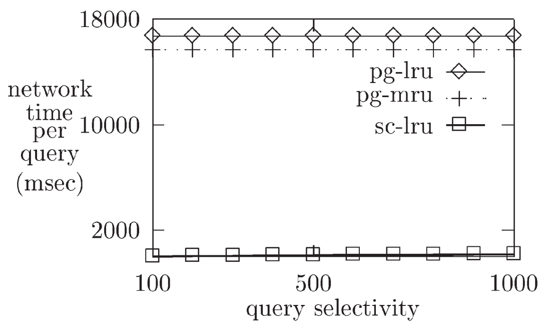


Fig. 18. SelAtt = A3, ProjIn = T, Skew = 0.8.

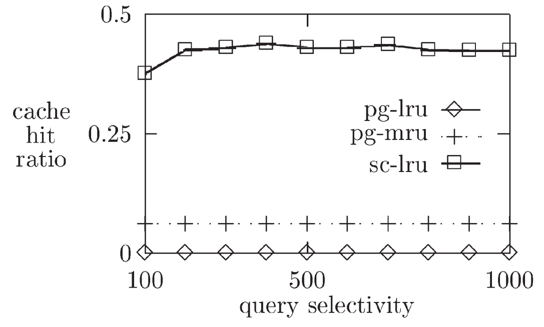


Fig. 19. SelAtt = A3, ProjIn = T, Skew = 0.8.

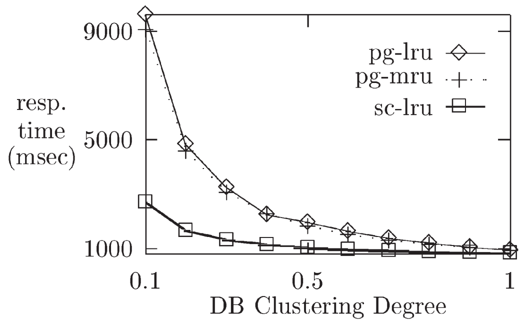


Fig. 20. SelAtt = A2, SelRag = 400, NumProjAtt = 50, ProjIn = T.

file scan. For each query requested, all the database pages must be scanned to compute the result. Therefore, the amount of data processed and transferred over the network is independent of the query size, as shown in Fig. 18. For semantic caching, under the assumption made in Section 5.1.2, the part of the query processing cost at the client side is very small compared to that at the server side, who also computes remainder queries by a file scan. Hence, the performance of semantic caching does not show much difference when the query selectivity varies.

When A3 is selected, semantic caching greatly outperforms page caching. At first, the cache hit rate of semantic caching is much higher than that of page caching (Fig. 19). When LRU is used, the page cache hit rate is close to zero. As each query visits all the database pages in sequential, when the cache is small, there is little chance for a cache hit. This fixed page access pattern also makes MRU a better policy, as shown in the figures. Second, the network traffic of semantic caching is much lower than those of page caching (Fig. 18). This is because too many useless data pages are send over the network.

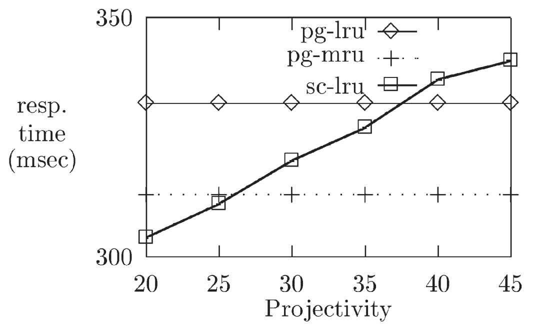


Fig. 21. SelAtt = A2, SelRag = 400, ProjIn = T, Skew = 1, BandWidth = 8Mb/s.

To further show that the performance of caching schemes is sensitive to the database physical organization, we study the performance by varying the degree of database clustering. Fig. 20 compares the cache schemes when the clustering degree is set from 0.1 to 1. It is shown that the difference decreases as the clustering degree increases. When the attribute is fully clustered, the two schemes perform similarly. While semantic caching behaves relatively stable under different scenarios, the performance of page caching is impacted by the clustering degree.

Though semantic caching outperforms page caching in most cases, there do exist situations when the performance of semantic caching is worse. Such a scenario occurs in Fig. 16, where the worse performance is caused by a coarse caching granularity. Fig. 21 shows another scenario. Fig. 21 illustrates the impact of query projectivity on the performance, where the indexed and fully clustered attribute A2 is queried. As shown in previous results, this is the case when page caching has the best performance. To mitigate the impact of the network bandwidth on the performance, we set *BandWidth* to 8Mb/s. We notice that the performance of page caching is not sensitive to the query projectivity, while the performance of semantic caching gets worse as the projectivity increases. After some point, it becomes worse than page caching. This is explained by what follows. In page caching, data is accessed and transferred at a page level. The change in projectivity will slightly impact the client side query processing cost, but will not impact the number of pages to be visited, which is a major cost. Thus, the overall page caching performance is not impacted by projectivity, which can also be seen from Table 5. For semantic caching, when query selectivity is fixed, the larger

TABLE 5
Average Cache Hit Ratio: Page versus Semantic Cache when Select on A2, SelRag = 400, ProjIn = True, Skew = 1, BandWidth = 8Mb/s

projectivity	pg-lru HitRate	pg-mru HitRate	sc HitRate
20	0.206	0.25	0.27
25	0.206	0.25	0.25
30	0.206	0.25	0.21
35	0.206	0.25	0.21
40	0.206	0.25	0.17
45	0.206	0.25	0.17

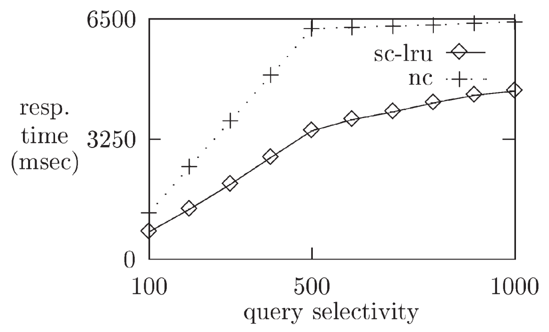


Fig. 22. SelAtt = A1, ProjIn = T, Skew = 0.8.

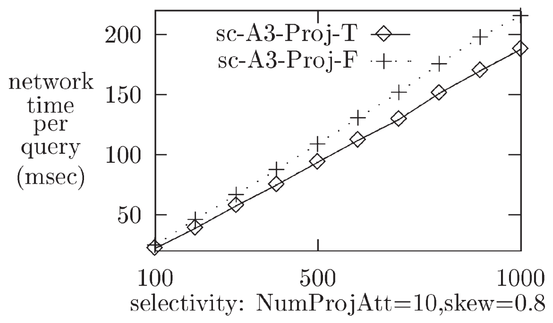


Fig. 24. SelAtt = A3, BandWidth = 1Mb/s.

the projectivity is, the bigger the query result is. Hence, less segments can be stored in the cache, which results in poorer cache hit rates (see Table 5) and worse response time. When the skew rate is high, say 1.0 in this experiment, there will be a large degree of overlapped pages between consecutive queries, which results in a higher page cache hit rate. However, according to the workload, there are only 10 hot attributes. Hence, each query generated will always involve some cold attributes, which decreases the degree of semantic locality among the workload and the semantic cache hit rate. When the hit rate of semantic cache is smaller than that of page cache, more data items need to be fetched from the network, which results in a worse response time. From this experiment, we notice that semantic caching is more sensitive to workloads and query types, which will be further examined in later experiments.

At last, we compare the performance of semantic caching with the case when there is no caching on the client side. Fig. 22 and Fig. 23 show the performance results in terms of response time and network traffic when A1 is selected and *BandWidth* is set to 1Mbps. We can see that the performance of semantic caching is much better. This demonstrates that semantic caching is more effective in improving response time and saving network traffic when the query workload exhibits a good degree of semantic locality.

In conclusion, we summarize that semantic caching has the following advantages over page caching:

- Since only the necessary data items are requested from the server, semantic caching is more effective in saving network traffic.
- Semantic caching is more effective in making use of cache space.
- Semantic caching is less sensitive to database physical organizations. It behaves robustly under different organization cases.

However, it has the following potential disadvantages:

- A coarse caching granularity may result in bad cache space usage.
- Semantic caching is sensitive to workloads and query types.

5.2.2 Impact of Query Workload on Semantic Caching

In this part, we study how the performance of semantic caching is impacted by the nature of query workload. As semantic caching outperforms page caching in most cases, the performance results for page caching are not shown in this study unless there is an exception.

ProjIn. Fig. 24 compares the semantic caching performance when *ProjIn* is set to *True* and *False*. It can be seen that the performance when *ProjIn* is set to *False* is worse. When the queried attribute is not projected, to process probe queries from semantic segments, amending queries are needed to bring the missing information from the server side, which adds extra network traffic. Since *AttrHot* is set to 0.2, there are 10 hot attributes in total. Also, we put A3 in the hot attribute set. When *NumProjAtt* is 10 and *ProjIn* is *False*, for every query generated, it will always involve at least one cold attribute. Hence, the cache hit rate is smaller than that when *ProjIn* is *True*. The parameter *ProjIn* has no impact on the performance of page caching, since queries are not computed by reasoning among predicates, they are processed either through a file scan or indexes instead. Even when *ProjIn* is set to *False*, semantic caching still outperforms page caching.

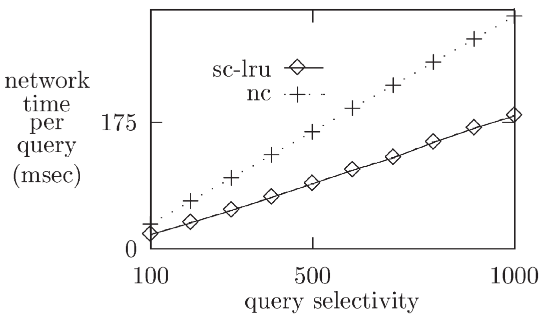


Fig. 23. SelAtt = A1, ProjIn = T, Skew = 0.8.

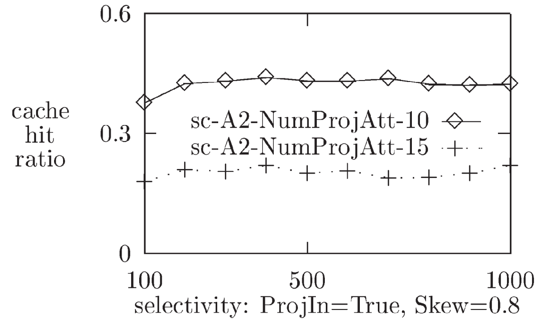


Fig. 25. SelAtt = A2, BandWidth = 1Mb/s.

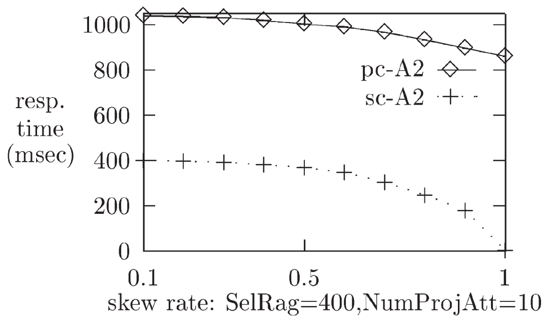


Fig. 26. SelAtt = A2, ProjIn = True, BandWidth = 1Mb/s.

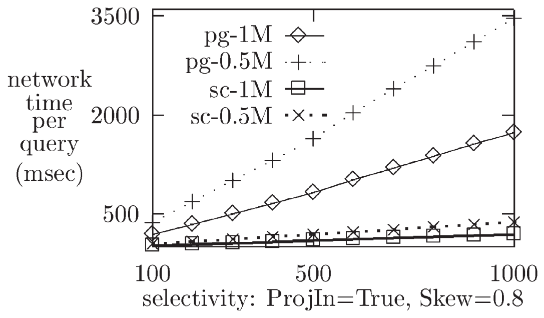


Fig. 27. SelAtt = A2, NumProjAtt = 10.

NumProjAtt. Fig. 25 compares the scenarios when the parameter *NumProjAtt* is set to 10 and 15. Since there are a total of 10 hot attributes, when *NumProjAtt* is set to 15, every query will contain at least part of cold attributes. If the skew rate is fixed, say 0.8, the degree of overlapping between consecutive queries when *NumProjAtt* is 15 is smaller than that when *NumProjAtt* is 10. In another words, the semantic locality among queries decreases. Therefore, the cache hit rate is reduced, the network traffic is increased, and the response time is worse.

Skew. Fig. 26 illustrates how the semantic cache performance is impacted when *Skew* changes from 0.1 to 1. Semantic caching outperforms page caching regardless of the skew rate.

In summary, we observe that the performance of semantic caching is impacted by the following factors.

- query type, which involves the issues such as whether the queries attribute is always projected, the number of attributes projected, etc.
- characteristics of query workload, i.e., degree of semantic locality

5.2.3 Impact of Network Bandwidth

The objective of this experiment is to demonstrate how the two schemes work under systems with different capacity. Since bandwidth is one of the most important constraints in mobile environments, we focus on studying its impact here. The performance results are illustrated in Fig. 27. Obviously, the change in bandwidth will not have any impact on the cache hit rates of both schemes, since neither the database physical organization nor the property of query workload is changed. However, we can see that the change of network bandwidth has a much bigger impact on page caching than on semantic caching. This is because, even the same amount of data is transferred over the network, it takes a longer time when the network is slower. Therefore, it is safe to say that semantic caching is more suitable to be used in the mobile computing environment where the network bandwidth is very limited.

6 Conclusions and Future Work

In this paper, the semantic caching scheme has been examined from the aspects of definition, organization, query processing, and performance evaluations. A formal semantic caching model is given in Section 3. The concepts of predicates, queries, semantic segments, and semantic caches are carefully defined. The semantic caching query processing strategies are studied in Section 4. Since a semantic cache is composed of a set of semantic segments, we view the query processing problem from two layers. The first layer deals with how to process a query from a semantic segment and the second layer deals with how to process a query from the whole cache. We present a data structure called the query plan tree, which provides a detailed plan about how to process a query from a cache. Detailed simulation models have been constructed to measure the performance of the semantic caching. The simulation results show that the semantic caching scheme is very effective and attractive. With many intrinsic advantages, it can be widely used in client-server environments, mobile computing, heterogeneous systems, web applications, etc.

For the future research, we intend to work along the following dimensions. First, we plan to extend our semantic cache model to include more complicated queries such as joins. This will increase the complexity in query processing and cache management. One way to deal with these problems is to let the algorithms for semantic caching be adaptive to the available resources as well as the query types. For example, since query trimming and coalescing may prove to be costly for the likewise limited resources of a mobile unit, the granularity of data stored can be dynamically changed. We can cache either the result of a complete select-project-join query or simply the result of the selection. The previous query optimization research work and algorithms can also be used to generate query processing plans.

Semantic cache replacement strategy is another important direction of future research. A traditional approach usually selects replacement victims using temporal locality. The unique characteristics of semantic caching provides us a way to choose replacement victims according to semantic locality, which is the property that if a query is requested, other queries semantically related to it are also likely to be requested in the near future. Our goal is to develop a replacement scheme which can first locate then utilize such locality among queries.

Semantic caching also brings new requirements in cache coherency control. Like traditional cases, a semantic segment can become obsolete because its tuples are updated. On the other hand, when some newly inserted/modified tuples satisfy the semantic description of a segment, they must be added to the segment even though they are not in it before. This also causes the segment to be out-dated. Another observation is that an invalid segment can be recomputed or incrementally maintained. In a word, a cache coherency control scheme which best fits semantic caching must be developed.

ACKNOWLEDGMENTS

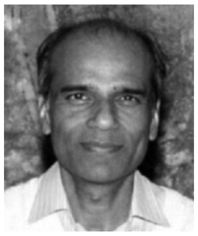
This work was supported partially by the US National Science Foundation (NSF) under grant number IIS-9979458. The authors are very grateful to Michael J. Franklin and Bjorn Thor Jonsson for many valuable discussions. We also thank the anonymous reviewers for their helpful suggestions that improved the quality of this paper.

REFERENCES

[1] B.T. Jonsson, "SIUMEI: Design Overview and Class Interfaces," http://www.cs.umd.edu/projects/dimsum/, 2000.
[2] S. Cluet, O. Kapitskaia, and D. Srivastava, "Using LDAP Directory Caches," *Proc. Symp. Principles of Database Systems,* 1999.
[3] C.M. Chen and N. Roussopoulos, "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching," *Proc. Int'l Conf. Extending Database Technology,* pp. 323-336, 1994.
[4] "CSIM18 Simulation Engine (C++ Version)," *Proc. ACM Conf. Information and Knowledge Management,* Mesquite Software Inc, 1996.
[5] S. Dar, M.J. Franklin, B.T. Jonsson, D. Srivatava, and M. Tan, "Semantic Data Caching and Replacement," *Proc. VLDB Conf.,* pp. 330-341, 1996.
[6] D.J. Dewitt, P. Futtersack, D. Maier, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proc. Conf. Very Large Databases,* pp. 107-121, 1990.
[7] M.H. Dunham and A. Helal, "Mobile Computing and Databases: Anything New?" *SIGMOD Record,* vol. 24, no. 4, pp. 5-9, Dec. 1995.
[8] P.M. Deshpande, K. Ramasamy, A. Shukla, and J.F. Naughton, "Caching Multidimensional Queries Using Chunks," *Proc. SIGMOD,* pp. 259-270, June 1998.
[9] M.J. Franklin, *Client Data Caching, A Foundation for High Performance Object Database Systems.* Kluwer Academic, 1996.
[10] P. Godfrey and J. Gryz, "Semantic Query Caching for Heterogeneous Databases," *Proc. KRDB Conf. Very Large Databases,* vol. 6, pp. 1-6, 1997.
[11] A. Gupta and I. Singh Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *Data Eng. Bull.,* vol. 18, no. 2, pp. 3-18, 1995.
[12] J. Gray and A. Reuter, *Transactional Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
[13] J. Gray, *The Benchmark Handbook.* Morgan Kaufmann, 1993.
[14] S. Guo, W. Sun, and M.A. Weiss, "Solving Satisfiability and Implication Problems in Database Systems," *ACM Trans. Database Systems,* vol. 21, no. 2, pp. 270-293, 1996.
[15] S. Guo, W. Sun, and M.A. Weiss, "On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 8, no. 4, pp. 604-616, 1996.
[16] T. Imielinski and H.F. Korth, "Introduction to Mobile Computing," *Mobile Computing,* pp. 1-43, Kluwer Academic, 1996.
[17] A.M. Keller and J. Basu, "A Predicate-Based Caching Scheme for Client-Server Database Architectures," *VLDB J.,* vol. 5, no. 2, pp. 35-47, Apr. 1996.
[18] D. Lee and W.W. Chu, "Semantic Caching via Query Matching for Web Sources," *Proc. CIKM,* pp. 77-85.
[19] K.C.K. Lee, H.V. Leong, and A. Si, "Semantic Query Caching in a Mobile Environment," *Mobile Computing and Comm. Rev.,* vol. 3, no. 2, pp. 28-36, Apr. 1999.
[20] P.A. Larson and H.Z. Yang, "Computing Queries from Derived Relations," *Proc. Conf. Very Large Databases,* pp. 259-269, 1985.
[21] R. Ramakrishnan, *Database Management Systems.* McGraw-Hill, 1998.
[22] D.J. Rosenkrantz and H.B. Hunt, "Processing Conjunctive Predicates and Queries," *Proc. Conf. Very Large Databases,* pp. 64-71, 1980.
[23] N. Roussopoulos, "An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis," *ACM Trans. Database Systems,* vol. 16, no. 3, pp. 535-563, 1991.
[24] X. Sun, N.N. Kamel, and L.M. Ni, "Processing Implication on Queries," *IEEE Trans. Software Eng.,* vol. 15, no. 10, pp. 1168-1175, 1989.
[25] J.D. Ullman, *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

Qun Ren received the MSc degree in computer science from Nanjing University, China, in 1991, and the PhD degree in 2000, from Southern Methodist University. Since 2000, she has been working as a system analyst with the Mobile Internet Applications division at Nokia Networks. Her research interests are semantic caching, query processing, mobile database, location dependent applications, WAP, and mobile computing.

Margaret H. Dunham received the BA and MS degrees in mathematics from Miami University, Oxford, Ohio, and the PhD degree in computer science from Southern Methodist University in 1970, 1972, and 1984, respectively. From August 1984 to the present, she has been first an assistant professor, an associate professor, and now a full professor in the Department of Computer Science and Engineering at Southern Methodist University in Dallas. Professor Dunham's research interests encompass main memory databases, data mining, temporal databases, and mobile computing. Dr. Dunham served as editor of the ACM SIGMOD Record from 1986 to 1988. She has served on the program and organizing committees for many ACM and IEEE conferences. She served as guest editor for a special section of *IEEE Transactions on Knowledge and Data Engineering* devoted to Main Memory Databases as well as a special issue of the ACM SIGMOD Record devoted to Mobile Computing in databases. She served as the general conference chair for the ACM SIGMOD/PODS held in Dallas in May 2000. She is currently an associate editor for *IEEE Transactions on Knowledge and Data Engineering*. She has published over seventy technical papers in such research areas as database concurrency control and recovery, database machines, main memory databases, and mobile computing.



Vijay Kumar is an associate professor of computer science in the School of Interdisciplinary Computing and Engineering, University of Missouri-Kansas City, Missouri. His research activities are in the areas of mobile computing, the Web, data warehousing, bioinformatics, e-commerce, and e-university. He has published papers in *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Computers*, and many other journals. At present, he is looking into mobile transaction modeling and management, mobile database recovery, and structure-functional properties of protein molecules for developing a query-based intelligent biological information provider. He is a member of ACM and IEEE.