Semantic Caching in Mobile Computing*

Qun Ren, Margaret H. Dunham
Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275

Abstract

Semantic caching is particularly attractive for use in a mobile computing environment due to the reduced wireless network traffic and the fact that it provides more autonomy for the mobile clients. 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 the performance of the semantic cache model is examined through a detailed simulation study, which shows its effectiveness in mobile computing.

Keywords: Semantic Caching, Query Processing, Query Trimming, Mobile Computing, Page Caching

1 Introduction

Since the first generation analog cellular systems were deployed in U.S. and Europe in the 1980s, the wireless communication technology has grown rapidly. Nowadays, more and more people carry portable computers and mobile phones to access information or communicate with each other anywhere at anytime, even while on the move. Both the technologies and applications lead to the emergence of a new computing paradigm: *mobile computing*. Compared to the well-known *distributed computing*, mobile computing exhibits unique characteristics due to the nature of the wireless communication media and mobility ([DH95], [IK96]). First of all, the bandwidth of wireless links that connect mobile units to the fixed network is very limited and therefore is a major performance bottleneck. While a wireless LAN can provide a bandwidth in the order of 10Mb/s, the typical bandwidth of wireless cellular systems is only in the order of 10kb/s. Disconnection is another distinguishing feature. Users may turn off their mobile computers to save battery energy, which is called voluntary/foreseeable disconnection; disconnection can be unpredictable too, e.g. as a result of wireless network failure. Furthermore, mobile units are usually resource poor devices, they often have relatively slow CPUs, limited memory/disk space and use short lifetime batteries as energy support.

Data caching at mobile units can reduce wireless communication and has been shown to be an effective way in handling the above problems ([CSL98], [LLS99], [CTO97]). Page caching is a traditional approach

*This material is based upon work supported by the National Science Foundation under Grant No. IIS-9979458

and is widely used in client-server systems. It intends to exploit static *spatial locality*[1], 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, which is not feasible during disconnections. In a word, traditional page caching is not an ideal choice for mobile computing.

The idea of *semantic caching* is that the mobile client maintains in the cache both semantic descriptions and results of previous queries ([DFJ96], [KB96]). 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 wireless link can be substantially reduced. To further illustrate the advantages of semantic caching in mobile computing, we compare it with page caching in Table 1.

Table 1: Semantic Caching vs. Page Caching in Mobile Computing

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
disconnection handling	efficient	inefficient	more autonomy of mobile clients & partial results may be derived in SC
coherency control	efficient	inefficient	out-dated data items can be incrementally maintained in SC

Since semantics about the cached items are stored in a semantic cache, the mobile 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 wireless 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 wireless link 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 makes better use of the mobile client cache space than page caching in many applications.

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 mobile client and the server work

[1] Here, spatial locality means that the data items which are physically stored closely are likely to be accessed together.

in parallel. By doing so, 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 ([DFJ96]). The missing pages are not pre-determined, instead, they are requested from the server during the query processing procedure.

The self-reasoning ability of semantic caching also facilitates disconnections as it makes a mobile client more autonomous. When the server is not available, the mobile client is still able to determine whether a query can be answered locally or not. Semantic caching offers benefits in cache coherency control too. Since cached semantic segments are regarded as materialized views, they can be maintained incrementally ([GM95], [Rou91]). That is to say, the changes of an invalid segment are computed by performing corresponding operations on the updated records rather than the entire base relation. Then only those changes are sent over the wireless network. By doing so, the cache investment can be effectively protected.

From what we have discussed so far, it can be seen that semantic caching exhibits many advantages in mobile computing. There has been much recent interest in this area. Previous work either ignores the formal definitions, or lacks precise query processing algorithms. Moreover, some of them only work in specific application domains, such as [DRSN98] and [CKS99], while others limit query types, such as [DFJ96] which focus on 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, by taking into account the characteristics of mobile computing. At last, we validate the performance of our semantic cache model through a detailed simulation study.

The rest of this paper is organized as following. 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 ([Rou91], [CR94]). While it is mainly used to reduce disk access in a centralized architecture, the semantic caching scheme in a client-server environment aims at alleviating the network contention and improving the system scalability ([DFJ96], [KB96]). Moreover, semantic caching is particularly attractive for use in mobile computing due to the fact of more autonomy of the mobile clients ([LLS99]). Applications in specific domains, such as OLAP ([DRSN98]), LDAP ([CKS99]), WWW ([LC99]) and heterogeneous systems ([GG97]), 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 [KB96], 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. [KB96] 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 [DFJ96] is also based on the queries executed. Unlike [KB96], they specify cached items individually. The semantic cache is defined as a collection of *semantic regions*, each having a constraint formula describing its content. [DFJ96] considers only select queries. Other semantic caching approaches have been proposed elsewhere, with [LLS99] in mobile computing, [DRSN98] in OLAP, [GG97] in heterogeneous systems and so on. However, most of the semantic cache papers to date haven't 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* ([DFJ96], [LLS99]), or as a number of pointers pointing to records of underlying relations needed to compute it ([Rou91], [CR94]). 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 ([CR94]). However, the pointer approach might not be profitable in a distributed system, especially in a low bandwidth mobile computing environment. [DFJ96] 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 [LLS99] 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 sub-regions (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, never coalescing approach leads to a large number of semantic segments, which may bring excessive overhead for query processing. The strategy used in [DFJ96] is to coalesce regions with the same cache replacement value into one region. Because they deal with only selections, this approach won't cause any difficulties in expressing the semantic meaning of the resulting region. [LLS99] 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 [KB96] 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 [DFJ96], which further calls the two split parts *probe query* and *remainder query* respectively. However, no detail query processing strategies are given in either work. [GG97] and [LLS99] categorize the possible relationship between a query and a cached fragment into several different scenarios. For every scenario, [LLS99] gives corresponding query processing strategies, but it does not provide the related theoretical base. [CKS99] 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 [DRSN98], 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.

Lots of works have been done in solving the satisfiability and implication problems ([RH80], [ULL89], [SKN89], [GSW96], [GSW+96]). 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 ([RH80], [SKN89]). 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 [LY85] and [CKS99].

Processing queries via a semantic cache is examined in [DFJ96], [CR94] and [KB96]. [DFJ96] works on only select queries, it treats all the candidate semantic regions as a whole to compute the corresponding probe and remainder queries. [CR94] works in a centralized system model and it mainly deals with optimization problems. A brief discussion is presented in [KB96] 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 total cache. Third, we further optimize the query processing procedure to parallelize the work between the server and the mobile 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

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 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 ([GM95]). 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. $P_j = b_{j1} \wedge b_{j2} \wedge ... \wedge b_{jl}$, each b_{jt} is a compare predicate.*

In Definition 2, S_R and S_A define the base 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. Meanwhile, 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, $< Q_R, Q_A, Q_P, Q_C >$.*

To make the formal semantic caching definition complete, we give the definition for a special semantic segment called the *Empty Semantic Segment*. An empty semantic segment consists of no tuples, and there aren't any base relations and attributes involved in the computation. When the system is first brought online, the cache can regarded as consisting of a set of empty semantic segments.

Definition 4 *A semantic segment, S, where $S = < NULL, \Phi, False, \Phi >$, is called an* **Empty Semantic Segment**.

Moreover, in order to define a cache and process queries, different kinds of relationships among segments and queries must be explored. Thus we have the following definitions. The *equivalent* relationship is used to detect redundant data, while the *related* relationships can be applied to query trimming.

Definition 5 *Semantic segments $S_1 = < S_{1_R}, S_{1_A}, S_{1_P}, S_{1_C} >$ and $S_2 = < S_{2_R}, S_{2_A}, S_{2_P}, S_{2_C} >$ are said to be* **Equivalent** iff

1. $S_{1_R} = S_{2_R}$ and
2. $S_{1_A} = S_{2_A}$ and
3. S_{1_P} and S_{2_P} are equivalent.

Definition 6 *Semantic segment $S = < S_R, S_A, S_P, S_C >$ and query $Q = < Q_R, Q_A, Q_P, Q_C >$ are said to be* **Related** iff $S_R = Q_R$.

Finally, we give the formal definition of a semantic cache in Definition 7. A semantic cache is defined as a set of distinct semantic segments, and is initialized as a set of empty semantic segments.

Definition 7 *A* **Semantic Cache,** *C, is defined as $C = \{ S_i \mid S_i$ is a semantic segment; S_i and S_j are not equivalent if $i \neq j$. $\}$*

[DFJ96] uses the term "semantic region" to refer to the individually stored areas in the cache. Their region is similar to S_C or the segment content but is restricted to be the result of a selection operation only. Keller and Basu in [KB96] did not provide formal definitions for the caching components, which we call "semantic segments" in our study.

3.2 Semantic Cache Organization

There are several different strategies to physically store semantic segments. [DFJ96] 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 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 informations 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 base relation S_R, the attribute set S_A, the selection predicate S_P;
- the pointer pointing to the first page that stores the segment;
- the timestamp indicating when the segment was last accessed 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 ≤ 28*;
- S_4: **Select** *Ename, Age* **From** *Employee* **Where** *Age > 45*;

Also suppose that the first pages of the four segments are 2, 4, 3, 8 respectively, and S_i was last visited at T_i, then the index is shown as Table 1.

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

4 Semantic Caching Query Processing

To answer a query from a semantic cache, the following steps must be taken. First, we check whether a query can be partially or totally answered by the cache. If yes, the locally available results are computed directly from the cache. In the case 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, techniques and algorithms for semantic caching query processing are intensively examined.

4.1 Theoretic Foundation

Definition 8 *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 relational algebra expressions F_1 and F_2 containing only project and select operations, with F_1 involving only attributes in S_A, such that $F_1(S_C) \neq \phi$, and $F_1(S_C) = F_2(Q_C)$. Furthermore, if $F_1(S_C) = Q_C$, we say Q is* **fully answered** *from S; otherwise, we say Q is* **partially answered** *from S.*

Definition 8 is derived from the concept of *Derivability* defined in [LY85]. From Definition 8, we know that the key issue to compute a query from a cached segment is to find the corresponding functions F_1 and F_2, and to make sure that F_1 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_1 can not be found in S. So, in Definition 8, we add an additional restriction on F_1. 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) ∧ (Salary > 50k). Obviously, the result of Q is totally contained in S_4, since every tuple which satisfies (Age > 50) ∧ (Salary >*

50k) will always satisfy (Age > 45), and the corresponding F_1 is $\pi_{Ename}\sigma_{(Age>50)\wedge(salary>50k)}$. However, Q can't be computed from S_4, because the attribute "Salary", which is used in F_1 to further qualify the resulting tuples, is not in S_4 after the projection.

Hence, when processing a query from a semantic cache, we must consider not only the predicate relationships between the query and the segments, but also the relationship among the attribute sets.

Definition 9 *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. $Q_{P_A} = \{\ a \mid a$ is an attribute, and a 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 and Q are related, $S_A \cap Q_A \neq \phi$, $Q_P \wedge S_P$ is satisfiable, and $Q_{P_A} \subseteq S_A$, then Q is answerable from S.*
2. *If S and Q are related, $Q_A \subseteq S_A$, $Q_P \Rightarrow S_P$, 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 $S' =< S_R, Q_A \cap S_A, Q_P \wedge S_P, S_C' >$. Clearly, from Definition 2, we have $S_C' = \pi_{Q_A \cap S_A}\sigma_{Q_P}(S_C)$ and we let $F_1 = \pi_{Q_A \cap S_A}\sigma_{Q_P}$. Since $Q_A \cap S_A \neq \phi$, $Q_P \wedge S_P$ is satisfiable, we have $S_C' = F_1(S_C) \neq \phi$. Moreover, because $Q_{P_A} \subseteq S_A$, we know all the attributes involved in F_1 are contained in S. We then construct a new query $Q' =< Q_R, Q_A \cap S_A, Q_P \wedge S_P, Q_C' >$. Again from the definition for queries, we have $Q_C' = \pi_{Q_A \cap S_A}\sigma_{S_P}(Q_C)$ and we let $F_2 = \pi_{Q_A \cap S_A}\sigma_{S_P}$. As S and Q are related, i.e. $S_R = Q_R$, by comparing the definitions of S' and Q', we know that $S_C' = Q_C'$, i.e. $F_1(S_C) = F_2(Q_C)$. Till now, we have found the corresponding F_1 and F_2. According to Definition 8, we can derive 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. Then we have $S_C' = \pi_{Q_A \cap S_A}\sigma_{Q_P}(S_C)$, and $F_1 = \pi_{Q_A \cap S_A}\sigma_{Q_P}$, which involves only the attributes contained in S. Since S and Q are related, we have $S_R = Q_R$; also as $Q_A \subseteq S_A$, 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. $F_1(S_C) = Q_C$. According to Definition 8, 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 10 classifies variables into two types: *unique* and *uniquely determined*, which helps to further examine the relationship among attributes.

Definition 10 *Let P be a boolean expression with variables $a_1, a_2, ...a_n, b_1, b_2, ...b_m$. The variables $a_1, a_2, ...a_n$ are said to be* **unique** *and $b_1, b_2, ...b_m$ are said to be* **uniquely determined** *by $a_1, a_2, ...a_n$, with respect to P, if the following condition hold:*

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

Definition 11 *Consider a semantic segment $S = < S_R, S_A, S_P, S_C >$. Let A be the set of unique attributes and B be the set of attributes uniquely determined by A, with respect to S_P. If $A \subseteq S_A$, we say S is an*

extendible semantic segment, *A is the* **unique set** *of attributes of S, $S_A \cup B$, 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 A with respect to S_P, if a tuple consisting of attributes in A 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 extendible 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, it is thus the unique attribute, and S_4 is an extendible segment. Clearly, "Salary" can be uniquely determined by "Ename". To form ${S_4}^+$, we can 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 extendable 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 and Q are related, ${S_A}^+ \cap Q_A \neq \phi$, $Q_P \wedge S_P$ is satisfiable, and $Q_{P_A} \subseteq {S_A}^+$, then Q is answerable by S^+.*
2. *If S and Q are related, $Q_A \subseteq {S_A}^+$, $Q_P \Rightarrow S_P$, 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. $\square$

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

Definition 12 *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_{P_A} is its predicate attribute set. Then we have:*

1. *If S and Q are related, $Q_P \wedge S_P$ is satisfiable, then Q is answerable by $S^+ =< S_R, S_A \cup Q_A \cup Q_{P_A}, S_P, {S^+}_C >$.*
2. *If S and Q are related, $Q_P \rightarrow S_P$, then Q can be fully answered by $S^+ =< S_R, S_A \cup Q_A \cup Q_{P_A}, S_P, {S^+}_C >$.*

Proof: Let us first prove that S is extendible. 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 extendible. Further let ${S_A}^+ = S_A \cup Q_A \cup Q_{P_A}$, clearly, ${S_A}^+$ is an extended attribute set of S, and $S^+ =< S_R, S_A \cup Q_A \cup Q_{P_A}, S_P, {S^+}_C >$ 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

Since a semantic cache consists of a set of semantic segments, we consider the query processing problem from two aspects. At first we look at the issue of 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 assume every cached semantic segment is key-contained, and only those related to a query are chosen as the candidates to compute it.

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

Figure 1 illustrates the possible relationships between a query Q and its related semantic 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 Figure 1, the query trimming strategies are very similar. Because S is key-contained, we use Lemma 3 to derive the query trimming mechanisms for each case. In most cases we don't need to construct a complete S^+ to compute a query, which will be discussed in detail later.

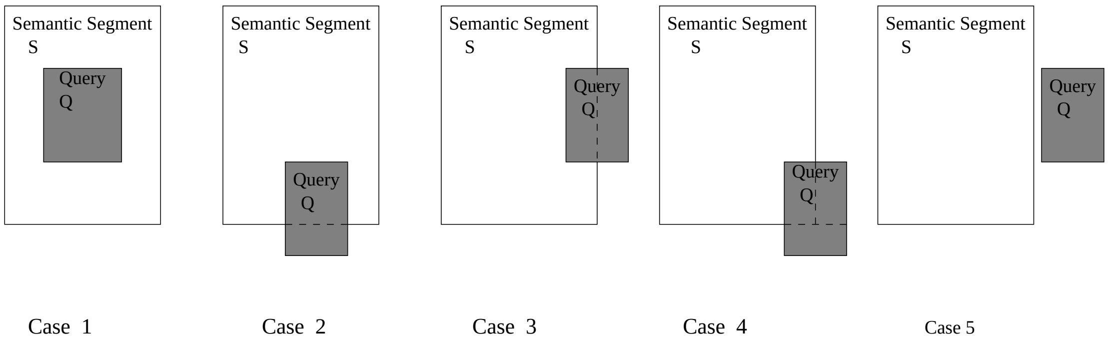

Figure 1: The Possible Relationships between Q and S

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 won't 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 AQ = $\pi_{K_A}\sigma_{Q_P}(Q_R)$ to fetch the corresponding key values of K_A from the server, and 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 $Q_A \subseteq S_A$ doesn't hold. Because S is key-contained, according to Lemma 3, we know that Q can be fully answered by S^+. As before, we don't 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,
- $A_2 = (Q_A \cap \neg S_A) \cup K_A$, 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 doesn't satisfy Q will be removed.

Case 4 - Hybridly Partitioned Here $Q_P \wedge S_P$ is satisfiable, but $Q_A \subseteq S_A$ doesn't 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 on base relations on server or further trimmed by other segments.

Since $Q_A \subseteq S_A$ doesn't 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 doesn't contain any result of Q. Since Q can not 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 related key-contained semantic segment S.

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; Q-related key-contained semantic segment S

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

Procedure: {

```
    K_A ← S's key attribute set;
    A_1 ← (Q_A ∩ S_A) ∪ K_A;  A_2 ← (Q_A ∩ ¬S_A) ∪ K_A;
    IF (Q_A ⊆ S_A) {
        IF (Q_P ⇒ S_P) {
            /***** Case 1: Totally Contained *****/
            *type = 1;
            IF ( Q_{P_A} ⊆ S_A ) THEN aq = NULL; ELSE aq = π_{K_A} σ_{Q_P}(Q_R);
            pq = π_{Q_A ∪ K_A} σ_{Q_P}(S_C);
            rq1 = rq2 = NULL; return; }
        IF ( Q_P ∧ S_P is satisfiable ) {
            /***** Case 2: Horizontally Partitioned *****/
            *type = 2;
            IF ( Q_{P_A} ⊆ S_A ) THEN aq = NULL; ELSE aq = π_{K_A} σ_{Q_P ∧ S_P}(Q_R);
            pq = π_{Q_A ∪ K_A} σ_{Q_P}(S_C);
            rq1 = π_{Q_A ∪ K_A} σ_{Q_P ∧ ¬S_P}(Q_R);
            rq2 = NULL; return; } }
    IF (Q_A ⊆ S_A) does not hold {
        IF (Q_P ⇒ S_P) {
            /***** Case 3: Vertically Partitioned *****/
            *type = 3; pq = π_{A_1}(S_C); rq1 = π_{A_2} σ_{Q_P}(Q_R);
            rq2 = aq = NULL; return; }
        IF ( Q_P ∧ S_P is satisfiable ) {
            /***** Case 4: Hybridly Partitioned *****/
            *type = 4;
            pq = π_{A_1}(S_C); rq1 = π_{Q_A ∪ K_A} σ_{Q_P ∧ ¬S_P}(Q_R);
            rq2 = π_{A_2} σ_{Q_P ∧ 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 combine 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 Figure 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.*

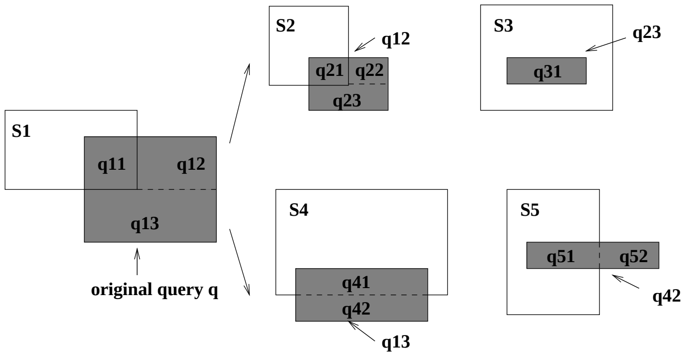

Figure 2: Example 3: A Semantic Cache Query Processing Example

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 attributes. The attribute "type" indicates whether that node is a leaf node, a "+" node or a "∨" node (The meanings of "+" and "∨" will be explained later). "subquery" gives the sub-query 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 "+" or "∨") 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 kids respectively. A query plan tree is created simultaneously with the query trimming procedure, which we will illustrate using Example 4.

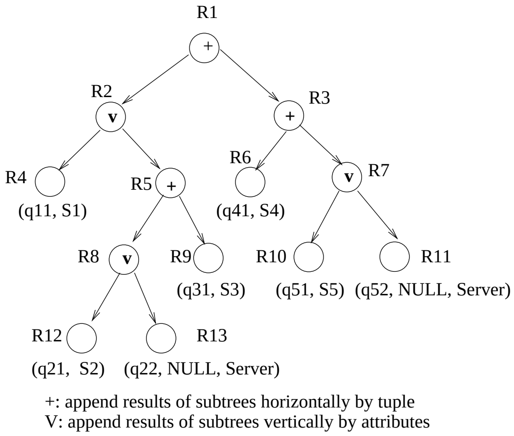

Figure 3: Example 4: A Query Plan Tree

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 Figure 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 "+", 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 "*$\vee$*", which indicates that the results of* R_2*'s subtrees are appended vertically by attributes. Till now, the left subtree of* R_2 *can't 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 exist several different ways to process a query. One straightforward approach, called *sequential query processing*, is to process each sub-query 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 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 via Example 5.

Example 5 *In this example, the action lists that are generated according to the query plan tree described*

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

Figure 4: Example 5: An ActionList Example

in Example 4 are illustrated in Figure 4. The server side action list includes all the subqueries that must be processed on the server side. The client side pre-action list contains the subqueries that can be computed without the help of the server; while the post-action list shows those subqueries that must wait for the results from the server to be processed. Suppose in Example 3, segment S_4 doesn't 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 post-action list. Obviously, the server can process the sub-queries in the server action list at the same time when the client works on the pre-action list. By doing so, the system performance is expected to be improved.

As we have indicated before, the cached semantic segments are disjoint 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, -1, srv_act, pre_clt, post_clt, plan).

Algorithm 2 Gen_Plan(Query *q, QueryPlan *parent, int direction, ActionList *srv_act, ActionList *pre_clt, ActionList *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 pre-action 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 a 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 "+". Create a new node "lgkid", associate it with the probe query pq and S. Let "lgkid" 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 "$\vee$". Create a new node "lgkid", associate it with the probe query pq and S. Let "lgkid" 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 "+". Create a node "lgkid", associate it with "$\vee$", and let it be the left child of "kid". Further create another node "lggkid", associate it with the probe query pq and S, and let it be the left child of "lgkid". 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, lgkid, "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 a 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 sub-query 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 "+" or "∨" node:
res1 ← pop (stack); res2 ← pop (stack);
IF (node = "+") THEN res ← append res1 and res2 by tuple
IF (node = "∨") THEN res ← append res1 and res2 by attribute
stack ← push (res); break;
Case a root node:
result ← pop(stack); break; } }

4.4 Coalescence and Decomposition

In this section, we discuss how to handle the scenario after a query is trimmed by a cached segment. Figure 5 shows such a typical scenario when a query Q is hybridly partitioned by a semantic segment S. Though there exist other cases such as totally contained, horizontally or vertically partitioned (refer to Figure 1), they are similar in principle.

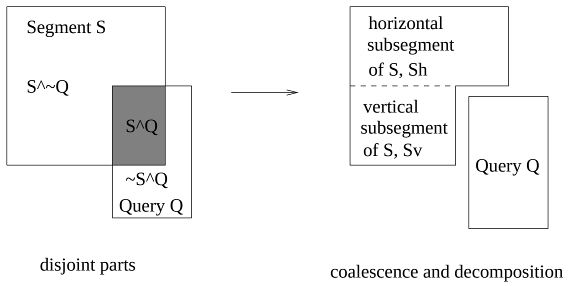

Figure 5: Handling Disjoint Parts After Query Trimming

According to Definition 7, a semantic cache should not contain redundant data. Hence, we can not keep the original segment S in the cache, and admit the result of Q as a new segment later on. One effective way to solve this problem is through decomposition and coalescence. As shown in the left part of Figure 5, three disjoint parts are left after query trimming. Here, we have the following choices:

- **Partial Coalescence in Query:** To decompose segment S into two parts: the overlapped part and non-overlapped part. Then to coalesce every part of Q together, i.e. to make the result of Q a new semantic segment.
- **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 found in the cache will be admitted.
- **No Coalescence:** To leave three disjoint parts separate.

The "no coalescence" approach can reflect 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. Obviously, the "partial coalescence in segment" approach could not effectively express the frequency of reference. The old part of the segment has a small chance to be replaced even though it is never been 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.

As shown in the right part of Figure 5, after the overlapped part is removed from S, the part which is left is irregular. That means, it is very difficult, if not impossible, to use a single semantic description to describe it. In order to treat and manage the decomposed part of S as semantic segments, we further divide it into two parts, namely, the *horizontal sub-segment* S_h and the *vertical sub-segment* S_v.

It is easy to find out that the definitions of these two segments can be generated at the same time as 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 sub-segment:** $S_h = < S_R, S_A, S_P \wedge \neg Q_P, S_{hC} >$.
- **vertical sub-segment:** $S_v = < S_R, (S_A \cap \neg Q_A) \cup K_A, S_P \wedge Q_P, S_{vC} >$.

5 Performance Study

We examine the performance of the proposed semantic caching model in mobile environments 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 ([CSIM18]), which simulates a simple but typical mobile computing model. In this section, we first discuss the simulation environment, then present the conducted experiments and further analyze the results.

5.1 Simulation Model

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

5.1.1 Mobile Client Model

Two types of mobile client models are simulated in this study: the page caching client and the semantic caching client. 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 which is implemented by the *Network Manager* module. However, 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 access the index and then access 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 wireless link. The space overheads for page cache management, such as Buffer Control Blocks ([GR93]), are simulated. At last, queries are processed by the *Query Processor* module.

For a semantic caching client, the semantic cache is managed by the *Semantic Cache Manager*. To maintain fairness, the space overheads for cache management, such as index, are also 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 semantic cache. To demonstrate the advantages of the parallel approach, the parallel as well as sequential query processing strategies are simulated. The part of a query which can not be answered locally will be sent to the server for processing.

5.1.2 Server Model

The server receives messages from the mobile client and processes them. A page caching client always sends a page request to the server, who then prepares that page and send it back to the client. To simplify the procedure, we assume that every page requested must be fetched from the disk. Things get a little bit more 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 these 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. On the other hand, 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, except that the buffer does not contain any useful information when the first subquery comes.

In summary, the server is composed of the following modules. The *Query Processor* processes queries from the mobile client. The *Buffer Manager* manages the memory buffer at a granularity of page. If a requested page can not be found in the buffer, it must be 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 mobile 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 mobile client CPUs adopt a FCFS scheduling policy, and 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 wireless network bandwidth is given by *BandWidth*. 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 [BTJ96]. The wireless 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, for more complicated operations such as predicate computation and reasoning, the costs can be easily derived. *HashCost* gives the CPU cost to do a hash function, which is used in page caching management.

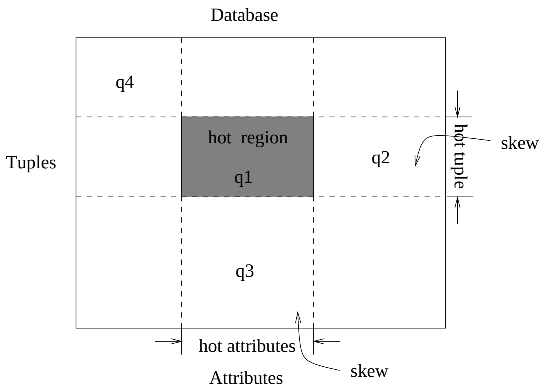

Figure 6: The Query Workload

We use a modified Wisconsin Benchmark ([Gray93]) 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 3 52-byte character strings. To compare the performance of semantic caching with page caching 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

Parameter	Description	Value
ServerMips	server CPU speed (Mips)	50
ServerCache	server side cache size (kb)	256
ClientMips	mobile client CPU speed (Mips)	50
ClientCache	client side cache size (kb)	128
BandWidth	wireless 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

Table 3: System and Workload Parameter Settings

indexed nor clustered. Moreover, we divide the database into two regions, namely, the *hot region* and the *cold region*. The size of the hot region is defined by *TupleHot* and *AttrHot* in Table 3. And the remainder part of the database is regarded as the cold region.

Queries used in this study involve only select and project operations. Also, 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 parameters related with queries are also listed in Table 3. The size of a query can be changed in the experiments and 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. Figure 6 further illustrates such a point. In this figure, query 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 of a query is bigger than the number of hot attributes, all the queries generated 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 after this point actually collect the performance metrics.

5.2 Experiments

The experiments conducted in this part are designed to compare the performance of page caching and semantic caching in mobile computing. Specifically, we explore how the following issues affect the performance:

- the database physical organization
- the nature of query workload
- the characteristics of the mobile system

We use the query response time as the primary performance metric, while other parameters such as wireless network traffic and cache hit ratio are also examined. The presented results are obtained by averaging the results of three runs of simulation. Each run consists of *WarmUpQry* queries to warm up the cache followed by *TotalQry* query executions when the actual measurements are taken.

5.2.1 Impact of Database Physical Organization

We examine two issues in database physical organization: indexing and clustering. As described in Section 5.1.3, the benchmark database R has three key attributes which differ in clustering and indexing. In what follows, we study the performance of the caching schemes when these attributes are queried respectively.

Figure 7, 8, 9 compare page with semantic caching when A1, which is an indexed but unclustered attribute, is queried. Also, we let A1 be always projected, namely, *ProjectIn* is set to "True". Meanwhile, *Skew* is 0.8 and *NumProjAtt* is 10. Moreover, we set the network bandwidth *BandWidth* to 1Mb/s. All the other parameters are the same as those shown in Table 3.

The x-axis of these figures represent the query selectivity, which ranges from 100 to 1000. When the value of *NumProjAtt* is fixed, the x-axis actually stands for the query size. The y-axis of Figure 7 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. Figure 8 gives how the network traffic changes as the query size increases. The y-axis represents the average wireless network time consumed per query. Finally, Figure 9 shows the metric of 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.

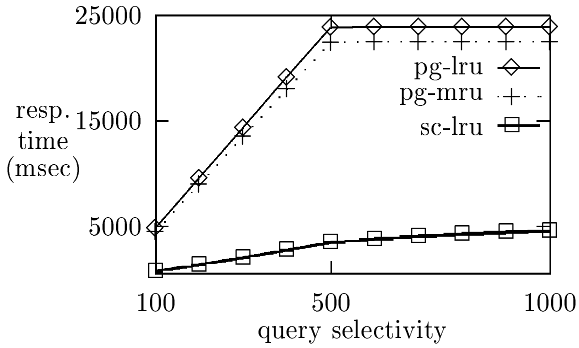

Figure 7: SelAtt=A1, ProjIn=T, Skew=0.8

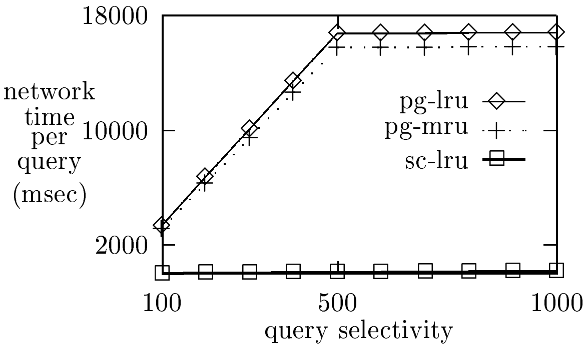

Figure 8: SelAtt=A1, ProjIn=T, Skew=0.8

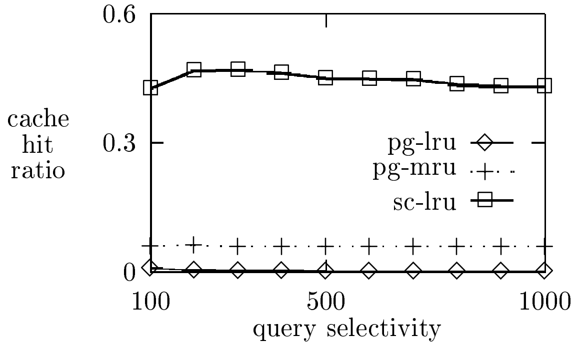

Figure 9: SelAtt=A1, ProjIn=T, Skew=0.8

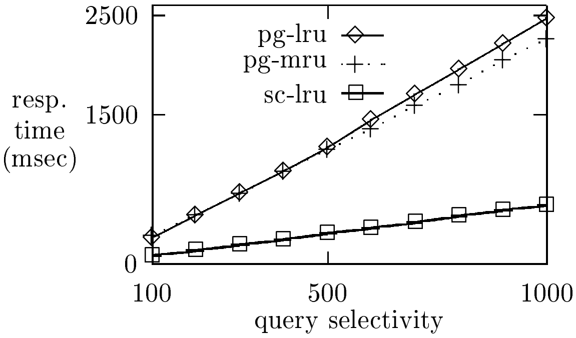

Figure 10: SelAtt=A2, ProjIn=T, Skew=0.8

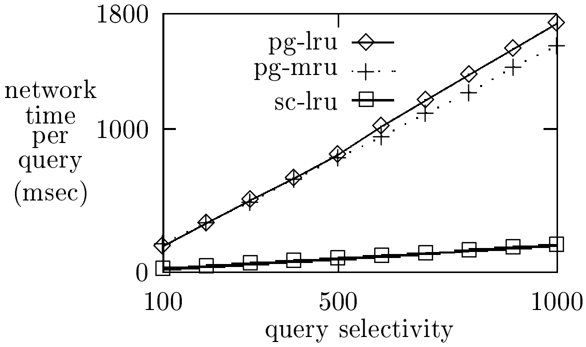

Figure 11: SelAtt=A2, ProjIn=T, Skew=0.8

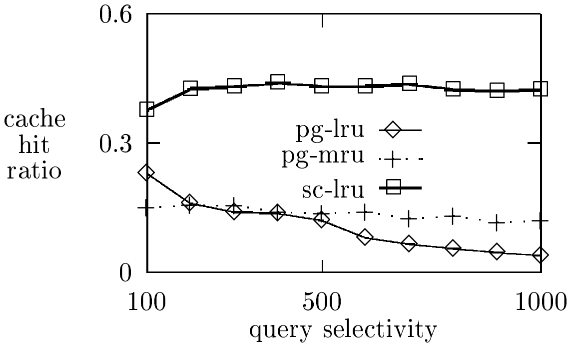

Figure 12: SelAtt=A2, ProjIn=T, Skew=0.8

In these figures, the performance of page caching is shown for two different replacement strategies: LRU and MRU. For semantic caching, the replacement policy LRU is used. From Figure 7, we can see that the response time for all the three parties worsens as the query size is increased. This is because more data has to be fetched from the server disk and transferred over the network, which can also be observed from Figure 8. 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 page 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 4096. For a totally unclustered attribute A1, when the selectivity is bigger than 500, the number of qualified pages to be accessed is always the same, i.e. 500. On the other hand, a semantic cache client sends the remainder queries to the server, who follows the same access path as the page caching client. That is why it shows a similar performance curve as page caching.

Also from these figures, we notice that semantic caching completely outperforms page caching. This is due to two reasons. First, as shown in Figure 9, the semantic cache hit rate is much higher than those of page caching. In a page caching case, to process a query, all the 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 (Figure 8). In page caching, a lot of unuseful data are sent from the server side.

Moreover, for page caching, MRU has a relatively better performance than LRU. 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 then LRU to keep those useful pages in the cache.

The set of figures Figure 10 to 12 illustrate the scenario when the selection is on attribute A2, which is both indexed and fully clustered. Similar to Figure 7, Figure 10 shows that the response time of all the three parties becomes worse when the query selectivity increases. However, unlike Figure 7, we don't have a special point at selectivity equals to 500. Instead, the response time worsens constantly. Because A2 is indexed, the number of qualified pages that a query needs to visit increases linearly as the selectivity increases. Further as A2 is fully clustered, the maximum number of qualified pages that a query might access, i.e. when selectivity equals to 1000, is less than the total pages of the database. Hence we won't have the case when different selectivity queries have the same number of qualified pages.

In this set of experiments, the page caching approach still performs worse than the semantic caching scheme. Moreover, the difference between the two schemes gets bigger as the query selectivity increases. In the page caching case, 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 page caching client still has to fetch a lot of unused attributes from the server, which causes lots of network traffic (Figure 11).

Also due to projections, semantic caching is more effective than page caching in utilizing the cache space. Hence its hit ratio is higher than that of a page cache (Figure 12). As the performance of semantic caching is mainly impacted by the nature of query workload, when *Skew* is fixed to 0.8, its hit ratio is relatively constant, as shown in Figure 12. When LRU is used, the hit ratio of page caching decreases as the selectivity increases. 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 Figure 9.

Overall, all the three approaches behave much better in this experiment than the previous one when A1 is queried. Thus whether the queried attribute is clustered or not impacts the performance of both page and semantic caching.

To further demonstrate the impact of projections on the performance, we study the scenario when *NumProjAtt* is set to 50, namely, all attributes are projected. Figure 13 illustrates the performance results in terms of response time. From Figure 10, we know that MRU performs better than LRU in page caching scenario, to make the figure clear, only the result of MRU is shown.

It can be seen in Figure 13 that the performance of semantic caching is very similar to that of page caching. If all the database attributes are projected, the advantages of semantic caching due to projections are lost. When *SelRag* is smaller 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 that are 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 won't 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 is impacted if the cache granularity is too coarse.

We now examine the performance of page and semantic caching when performing selections on A3, an unindexed and unclustered attribute. Figure 14, 15 and 16 illustrate the performance results in terms of response time, network time and cache hit ratio.

From these figures, we notice that all the three approaches behave constantly as the query selectivity increases. In a page caching case, since A3 is unindexed, the query access path is simply a 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 Figure 15. For semantic caching, under the assumption made in Section 5.1.2, the part of the query processing cost at

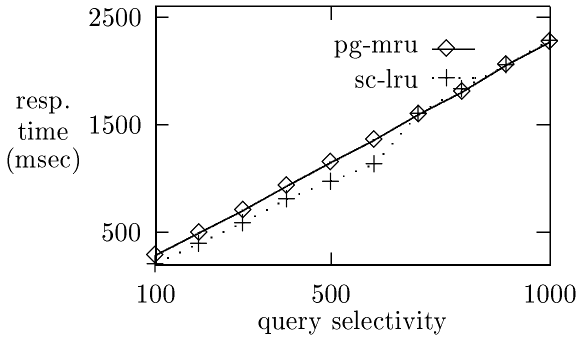

Figure 13: SelAtt=A2, NumProjAtt=50, Skew=0.8

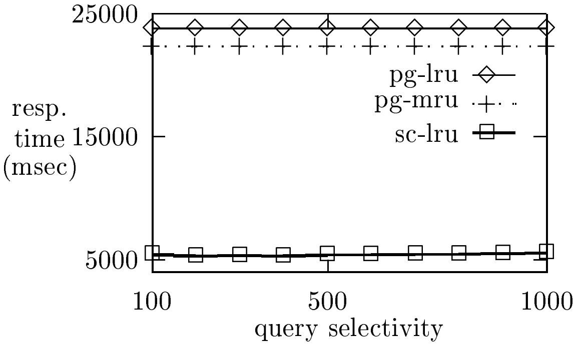

Figure 14: SelAtt=A3, ProjIn=T, Skew=0.8

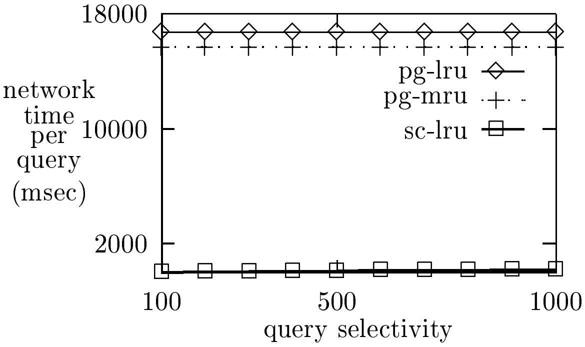

Figure 15: SelAtt=A3, ProjIn=T, Skew=0.8

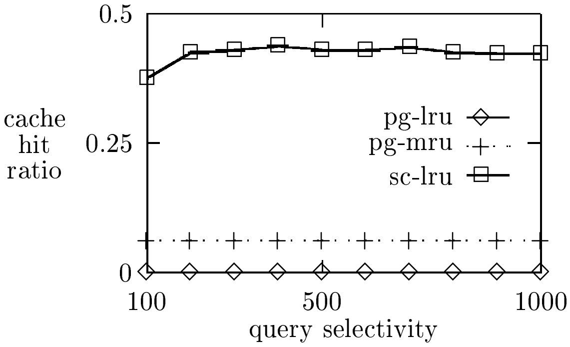

Figure 16: SelAtt=A3, ProjIn=T, Skew=0.8

the client side is very small compared to that at the server side, who also computes remainder queries by a file scan. Therefore, the performance of semantic caching does not show much difference when the query selectivity varies.

In this case, i.e. when A3 is selected, semantic caching greatly outperforms page caching. This is due to the following reasons. At first, the cache hit ratio of semantic caching is much higher than those of page caching (Figure 16). When LRU is used, the page cache hit ratio is close to 0. 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. Secondly, the network traffic of semantic caching is much lower than those of page caching (Figure 15). This is because too many useless data pages are send over the network.

Through comparing the three sets of experiments conducted so far, namely, selecting on A1, A2 and A3, we can see that the case when A3 is selected has the worst performance. In the scenario when A1 is selected, it also shows the worse performance after the selectivity reaches 500. Since A1 is totally unclustered, after the selectivity is or greater than 500, the number of pages that each query visits equals to the number of the total database pages. Moreover, clearly, the performance of caching schemes is sensitive to the database physical organization, which will be more carefully investigated in later experiments.

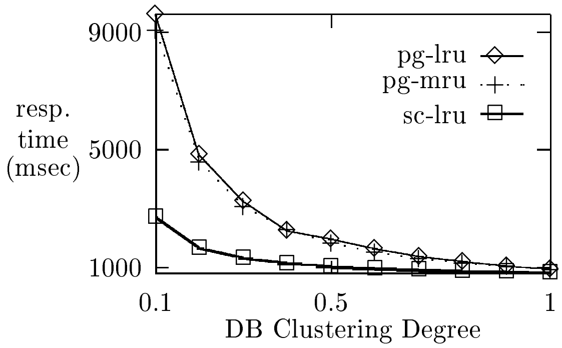

Figure 17: SelAtt=A2, SelRag=400, NumProjAtt=50, ProjIn=T

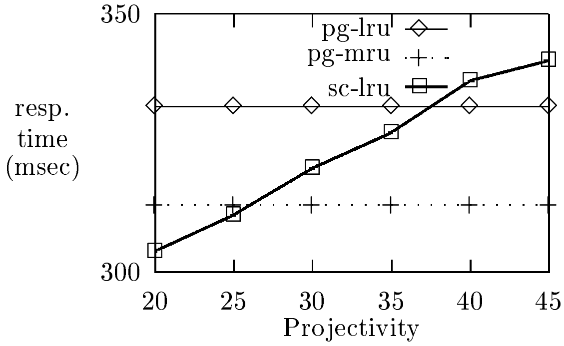

Figure 18: SelAtt=A2, SelRag=400, ProjIn=T, Skew=1, BandWidth=8Mb/s

To further demonstrate that page caching is more sensitive to database physical organization than semantic caching, we study the performance by varying the degree of database clustering. Figure 17 compares the cache schemes when the clustering degree is set from 0.1 to 1. From this figure, we can see that the difference between page and semantic caching 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 could be drastically impacted the clustering degree.

Table 4: Avg. Cache Hit Ratio: Page vs. 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

From the experiments studied so far, we notice that semantic caching outperforms page caching in most cases. Now let us try to find out the situations when the performance of semantic caching is worse. Such a scenario occurs in Figure 13, where the bad performance of semantic caching is caused by a coarse caching granularity. Figure 18 shows another scenario.

Figure 18 illustrates the impact of query projectivity on the performance. Here, the indexed and fully clustered attribute A2 is queried. As we know from previous results, this is the case when page caching has the best performance. Also, to mitigate the impact of the network bandwidth on the performance, we set *BandWidth* to 8Mb/s. From this figure, we can see 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. Moreover, 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 won't impact the number of pages to be visited, which is a major cost in that case. Thus the overall page caching performance is not impacted by projectivity, which can also be seen from Table 4. On the other hand, in semantic caching, when query selectivity is fixed, the larger 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 4) and worse response time. Moreover, 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 attribute, 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.

In conclusion, from these experiments, we can 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.

Also, 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

As we have indicated before, the performance of semantic caching is impacted by the nature of query workload. In what follows, we study such impacts by changing the workload along three dimensions. Because semantic caching outperforms page caching in most cases, the performance results for page caching won't be shown in this study unless there is an exception.

ProjIn Figure 19 compares the semantic caching performance under the situations when *ProjIn* is set to *True* and *False*. It can be seen that the performance when *ProjIn* is set to *False* is worse than that when *ProjIn* is set to *True*. This is due to the following reasons. As we have discussed in Section 4.2, when the queried attribute is not projected, in order 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. Moreover, in this simulation study, *AttrHot* is set to 0.2, i.e. there are 10 hot attributes in total. Meanwhile,

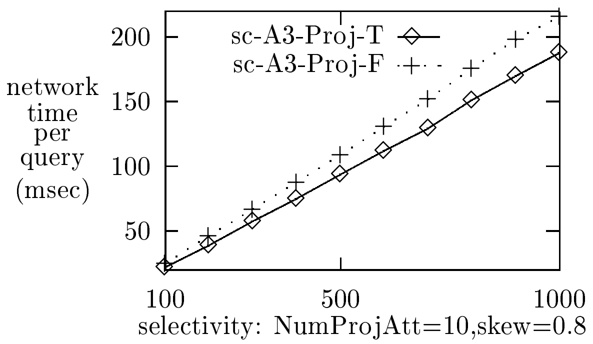

Figure 19: SelAtt=A3, BandWidth=1Mb/s

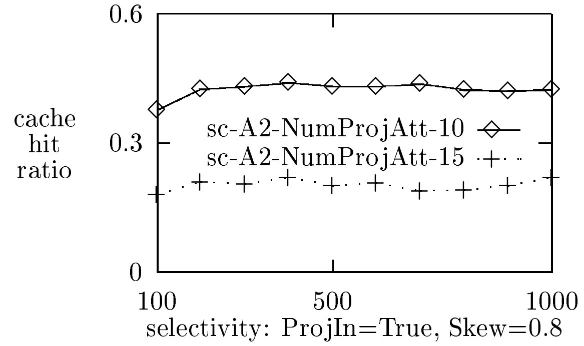

Figure 20: SelAtt=A2, BandWidth=1Mb/s

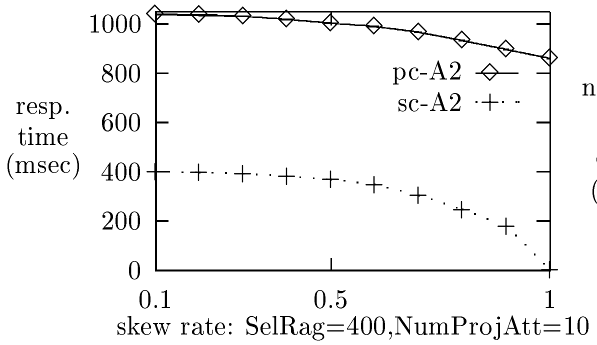

Figure 21: SelAtt=A2, ProjIn=True, BandWidth=1Mb/s

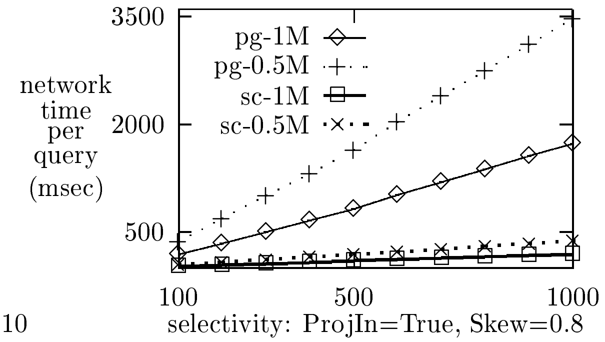

Figure 22: SelAtt=A2, NumProjAtt=10

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. Therefore, the cache hit rate of this case is smaller than that when *ProjIn* is *True*. However, the parameter *ProjIn* has no impact on the performance of page caching. In page caching, queries are not computed by reasoning among predicates, instead, they are processed either through a file scan or indexes. Also, even when *ProjIn* is set to *False*, semantic caching still outperforms page caching.

NumProjAtt Figure 20 compares the scenarios when the parameter *NumProjAtt* is set to 10 and 15. As we have discussed earlier, there are totally 10 hot attributes. When *NumProjAtt* is set to 15, every query issued will always contain at least part of cold attributes. If the skew rate is fixed, say 0.8, the degree of overlapping between consecutive queries in the case 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 Finally, we take a look at how the *Skew* rate impacts the semantic caching performance. Figure 21 illustrate the results when *Skew* changes from 0.1 to 1. Semantic caching outperforms page caching regardless of the skew rate.

In summary, from these experiments, 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, and so on
- characteristics of query workload, i.e degree of semantic locality

5.2.3 Impact of System

In previous experiments, we set *BandWidth* to 1Mb/s, and show that semantic caching outperforms page caching. The objective of this experiment is to demonstrate how the two schemes work under systems with different capacity. Since bandwidth is one of most important constraints in mobile environments, we focus on studying its impact here. The performance results are illustrated in Figure 22. Obviously, the change in the bandwidth won't 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, from the figure, 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.

5.2.4 Summary

The simulation results reported in this part show that in most cases, semantic caching exhibits much better performance than page caching in terms of response time, network traffic and cache hit rate.

Since it is known that page caching is quite sensitive to the database physical organization, we first study the two caching schemes along this dimension. Different indexing and clustering scenarios are examined in the experiments. Page caching is absolutely a loser when an unindexed attribute is queried. When the queried attribute has an index, the performance of page caching is impacted by the degree of clustering. The performance improves as the clustering degree increases, since less data pages need to be visited. In this case, semantic caching still outperforms page caching when the query projectivity is not so high. However, when all the attributes are projected, the two schemes behave similarly as the advantage of semantic caching due to projections are lost. In the extreme case when the query size is bigger that the cache size, semantic caching is worse than page caching since no data is admitted to the cache. It can be seen that the possibly coarse granularity could be a potential problem for semantic caching.

While semantic caching performs robustly under different database physical organization, it is sensitive to the nature of query workload. Its performance worsens when the selected attribute is not projected. To answer a query from the cache, amending queries are needed to bring the missing attributes from the server, which results in extra network traffic. Moreover, the semantic caching performance also depends on the semantic locality in the workload. This can be observed from two aspects. Semantic caching performs better when the skew rate is higher. The query results are more likely to overlap with each other when more queries visit the hot region. Also, if we increase the number of projected attributes so that all the queries involve at least some cold attributes, the degree of semantic locality is decreased. Therefore, the cache hit rate is reduced, the network traffic is increased, and the response time worsens. In the case when an indexed and fully clustered attribute is queried, which is the best situation for page caching, semantic caching is worse than page caching if its hit rate is lower. However, due to the way it processes queries, page caching is not impacted by query types in terms of projectivity and whether the queried attribute is always projected.

From the experiments conducted, we notice that semantic caching is more effective than page caching in reducing network traffic and making use of cache space. Hence it is a very attractive scheme for mobile computing where the network bandwidth is a severe constraint.

6 Conclusions and Future Work

In a mobile computing environment, mobile clients and database servers communicate via wireless channels that have limited bandwidth and are vulnerable to disconnection. Caching of frequently accessed data at the mobile client side can reduce the interactions between the client and server, therefore is very useful in handling these problems. In this paper, the semantic caching scheme has been thoroughly examined from the aspects of definition, organization, query processing and performance evaluations.

A formal semantic caching model has been given in Section 3. The concepts of predicates, queries, semantic segments and semantic caches are carefully defined. This work lays the theoretical foundation for the whole research. The semantic caching query processing strategies have been 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, which is also called query trimming. The theoretic foundations as well as the precise algorithms have been proposed, which are mainly built on the previous works in query folding and predicate reasoning. The second layer deals with how to process a query from a whole semantic cache. In this part, we have presented a data structure called the query plan tree, which actually provides a detailed plan about how to process a query from a cache. With the works done in these two layers, we have solved the semantic caching query processing problem in the select-project query domain.

Detailed simulation models have been constructed to measure the performance of the semantic caching in mobile computing. The simulation results show that semantic caching completely outperforms page caching in mobile computing, since it is more effective in saving wireless network traffic and making use of cache space. While page caching is quite sensitive to database physical organizations, semantic caching behaves robustly under different organization scenarios.

In summary, we have shown that the semantic caching scheme is a very effective and attractive way to do caching. With many intrinsic advantages, it can be widely used in client-server environments, mobile computing, heterogeneous systems, web applications and so on. Hence, it is safe to say that this is an interesting and valuable research topic.

For the future research, we intend to work along the following dimensions. First, we plan to extend our semantic cache model by including other more complicated queries such as joins. This will definitely increase the complexity in query processing and cache management. One possible 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 select part. Also, we need to consider the optimization issues in query processing and cache organization in the future. A mechanism to always find the most cost-effective way of query processing is expected to be developed. Moreover, we have to decide whether it is beneficial to keep a decomposed segment in the cache if its size is small. Finally, aspects such as cache coherency control, cache replacement strategies, admission and prefetching policies will be explored by combining the unique characteristics of semantic caching and mobile computing.

Acknowledgements

This work was supported partially by the National Science Foundation (NSF) under Grant No. IIS-9979458. The authors are very grateful to Michael J. Franklin and Bjorn Thor Jonsson for many valuable discussions.

References

[BTJ96] Bjorn Thor Jonsson, University of Maryland, *SIUMEI: Design Overview and Class Interfaces*, http://www.cs.umd.edu/projects/dimsum/

[CKS99] Sophie Cluet, Olga Kapitskaia and Divesh Srivastava, *Using LDAP Directory Caches*, Proceedings of PODS, 1999,

[CR94] Chungmin M. Chen and Nicholas Roussopoulos, *The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching*, Proceedings of the EDBT, 1994, pp. 323-336

[CSIM18] Mesquite Software Inc., *CSIM18 Simulation Engine (C++ Version)*

[CSL98] Boris Y. Chan, Antonio Si and Hong V. Leong, *Cache Management for Mobile Databases: Design and Evaluation*, Proceedings of ICDE, 1998, pp. 54-63

[CTO97] Jun Cai, Kian-Lee Tan, Beng Chin Ooi, *On Incremental Cache Coherency Schemes in Mobile Computing Environments*, Proceedings of ICDE, 1997, pp. 114-123

[DFJ96] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivatava and Michael Tan, *Semantic Data Caching and Replacement*, Proceedings of the VLDB Conference, 1996, pp. 330-341

[DH95] Margaret H. Dunham and Abdelsalam Helal, *Mobile Computing and Databases: Anything New?*, SIGMOD Record, Vol. 24, No. 4, December 1995, pp. 5-9

[DRSN98] P.M.Deshpande, K.Ramasamy, A.Shukla and J.F.Naughton, *Caching Multidimensional Queries Using Chunks*, Proceedings of SIGMOD, June 1998, P.259-270

[GG97] Parke Godfrey and Jarek Gryz, *Semantic Query Caching for Heterogeneous Databases*, Proceedings of KRDB at VLDB, 1997, pp. 6.1-6.6

[GM95] Ashish Gupta and Inderpal Singh Mumick, *Maintenance of Materialized Views: Problems, Techniques, and Applications*, Data Engineering Bulletin, Vol.18, No.2, 1995, pp. 3-18

[GR93] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993

[Gray93] Jim Gray, *The Benchmark Handbook*, Morgan Kaufmann Publishers Inc., 1993

[GSW96] Sha Guo, Wei Sun and Mark A. Weiss, *Solving Satisfiability and Implication Problems in Database Systems*, ACM Transactions on Database Systems, Vol.21, No.2, 1996, pp. 270-293

[GSW+96] Sha Guo, Wei Sun and Mark A. Weiss, *On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems*, IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.4, 1996, pp. 604-616

[IK96] Tomasz Imielinski and Henry F. Korth, *Introduction to Mobile Computing*, Mobile Computing, pp. 1-43, Kluwer Academic Publishers, 1996

[KB96] Arthur M. Keller and Julie Basu, *A predicate-based caching scheme for client-server database architectures*, The VLDB Journal, Vol. 5, No. 2, Apr. 1996, pp. 35-47

[LC99] Dongwon Lee and Wesley W. Chu, *Semantic Caching via Query Matching for Web Sources*, In Proceedings of CIKM, pp. 77-85

[LLS99] Ken. C.K.Lee, H.V.Leong and Antonio Si, *Semantic Query Caching in a Mobile Environment*, Mobile Computing and Communications Review, Vol. 3, No. 2, April 1999, pp. 28-36

[LY85] P.A.Larson and H.Z.Yang, *Computing Queries from Derived Relations*, Proceedings of VLDB conference, 1985, pp. 259-269

[RH80] D.J.Rosenkrantz and H.B.Hunt, *Processing Conjunctive Predicates and Queries*, Proceedings of VLDB, 1980, pp. 64-71

[Rou91] Nicholas Roussopoulos, *An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis*, ACM Transactions on Database Systems, Vol. 16, No. 3, 1991, pp. 535-563

[SKN89] X.Sun, N.N.Kamel and L.M.Ni, *Processing Implication On Queries*, IEEE Transactions On Software Engineering, Vol.15, No.10, 1989, pp. 1168-1175

[ULL89] J.D.Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1989