TECHNICAL REPORT 98-CSE-04

Semantic Caching and Query Processing

Qun Ren

Margaret H. Dunham

Department of Computer Science and Engineering

Southern Methodist University

Dallas, TX 75275–0122

qren@seas.smu.edu

May 26, 1998

Contents

List of Figures

List of Tables

Abstract

Data Caching at local clients plays an important role in improving the performance of client-server systems. In this paper, we introduce a novel caching scheme which is called semantic caching. Unlike the traditional page caching, tuple caching etc., the semantic cache is defined and accessed based on the queries submitted by the client. Both the predicates and the resulting tuples of queries are cached. When a new query is requested, the partial or whole result may be obtained through query processing on the semantic cache. With the advantages of low overhead, efficiency and flexibility, the semantic caching scheme can be used in the mobile computing environment,heterogeneous systems and general client-server systems. The formal definition of the semantic cache and related concepts are first given in this report, we then discuss the semantic cache management issues. At last, we investigate the semantic caching query processing techniques.

Keywords: Semantic Cache, Semantic Segment, Query, Query Trimming, Coalescing, Cache Replacement, Satisfiability, Implicity.

1 Introduction

Traditional caching schemes intend to exploit the static spatial or temporal locality to improve system performance and therefore have inherent disadvantages. For example, the performance of page level caching is extremely sensitive to teh way data is clustered into pages. In this study, we investigate the semantic locality of the queries submitted by the client and propose a novel caching scheme - *semantic caching*. The client maintains in the cache semantic descriptions and related results of previous queries. When a new query comes, if it can be answered from the cache, then no interaction with the server is needed; if it can be partially answered by the cache, then the original query is trimmed and the trimmed part is processed on the server. By processing queries in this way, the amount of data transferred between the client and the server can be greatly reduced.

The efficiency of the semantic caching scheme depends on how much the queries submitted by the client are related to each other semantically. We find a lot of such kind of applications. For example, a user who invests stocks may want to inquire the stock prices. The query series he submits are more likely to be semantically related, namely, the results of earlier queries are contained in the results of later queries. Another example for high-semantically related query series is a WWW search. People tend to do a sequence of searches on similar or related subjects.

One key advantage for the semantic caching is that the memory or disk space requirements and communication costs are reduced, because only the data that satisfy a given query are transferred to and stored on the client side. This advantage is very important in the mobile computing environment, as the wireless link and mobile unit memory, disk and CPU resources are very limited. In addition, when the wireless link is disconnected, if a query can be partially computed from the cached contents, at least some results may be returned to the user. Another advantage of the semantic caching is its flexibility, which is very important for heterogeneous environments. Tuple-cache and page-cache schemes are relatively inflexible, because the contents of the cache are dependent on the physical storage structure. Furthermore, it is not clear what tuple caching or page caching across heterogeneous databases even means([GG97]).

The remaining sections are organized as follows: In Section 2 we review previous research which is related to our semantic caching scheme. Section 3 gives a formal definition of the semantic caching scheme and defines the semantic caching model. Issues of semantic caching management are discussed in Section 4. Section 5 investigates the semantic caching query processing strategies. Finally, we draw conclusions in Section 6.

2 Related Research

In this section, the research related to our semantic caching study are reviewed. At first, we discuss *query folding*, which is a technique for determining if and how a query can be answered using a given set of derived relations. Then works on caching mechanisms in client-server environments are also reviewed.

2.1 Query Folding

Query folding refers to the activity of determining if and how a query can be answered using a given set of resources, which might be materialized views, cached results of previous queries, or queries answerable by other databases([Qia96]). In this section, research work and results in query folding and related *satisfiability* and *implication* problems are given.

The task of query folding is to try to find a rewriting of a query using the set of resources. All the possible rewritings can be obtained by considering *containment* mappings from the sources to the query. [LMS95] shows that the problem of finding rewritings that mention as few of the database relations as possible is NP-complete for conjunctive queries and resources. There are two independent sources of complexities in query folding: the first comes from the number of possible mappings from the resources to the query, and the second is to determine which parts of the query can be removed when the mapped resources are added to the query([LMS95]).

A lot of research work has been done in query folding([LY85], [YL87], [CKPS95], [LMS95], [Qia96], [Gry98], etc.) In the earlier work of Yang and Larson [LY85], they gave necessary and sufficient conditions for when a query is computable from a single derived relation; in [YL87], they considered the problem of finding rewritings for conjunctive queries using the available set of derived relations. Their approach considers what amounts to one-to-one mappings from the derived relations to the query, and does not search the entire space of rewritings, therefore may not always find all the possible rewritings of the query. [CKPS95] finds the rewritings for SPJ queries such that the rewritten query preserves the *bag semantic*, i.e. if it results in the same set of tuples. [CKPS95] also considered the question of how to extend a query processor to choose between the different rewritings. Levy et al. [LMS95] first gave a conclusion that query folding when both the query and the set of derived relations are conjunctive queries is NP-complete. Then, they developed a polynomial time algorithm to determine which set of redundant literals can be removed from the query. Under certain conditions there is a unique maximal set of such literals and the algorithm is guaranteed to find them. [Qia96] developed an exponential-time algorithm that finds all complete or partial foldings by using the folding rules derived from the set of resources. A polynomial-time algorithm for the subclass of acyclic conjunctive queries is also given in [Qia96]. Gryz, in his work [Gry98], takes integrity constraints into account in query folding. He described a complete strategy for finding foldings in the presence of inclusion dependencies and presented a basic algorithm that implements that strategy, the algorithm is also extended to consider both inclusion and functional dependencies.

Checking the satisfiability, equivalence or implication relationship between the query and the set of derived relations is the first step for query folding. Comparing a derived relation predicate P and a query predicate Q, there are three scenarios:

- P∧Q is unsatisfiable, implying that the derived relation will not contribute to the answer of the query Q;
- Q→P, i.e. Q implies P, implying that the whole answer of Q is contained in the derived relation;
- Neither P contradicts Q nor Q implies P, implying that there may exist tuples in the derived relation that contribute to the answer of the query Q.

The satisfiability and implication problem, and the closely related problem of query equivalence, have been solved for certain classes of queries: conjunctive queries involving arithmetic inequalities. We summarize the work in this area in Table 1. Here, $OP_{\neg\neq}$ stands for the operator set $\{=, <, \leq, >, \geq\}$, without the operator $\neq$; OP_{all} stands for the operator set including all the six operators. $|S|$, $|T|$ stand for the number of comparisons in predicates S and T respectively, n is the number of variables in S.

The satisfiability and implication problems have been studied intensively in the literature([RH80], [ULL89], [SKN89], [GSW96], [GSW+96]), just as Table 1 shows. Notice that, when $\neq$ comparison between variables is allowed, the problems become more difficult, or even NP-hard in the integer domain.

2.2 Caching Issues

The caching of data at client workstations has proved to be an effective technique for improving the performance of a client-server database system ([CFL91], [FC94], [Fra96]). By caching data for later reuse, the client-server interactions can be reduced, thus improving the response time and lowering the network traffic. Because queries can be processed based on the data cached at the client side, the client CPU, memory and

Table 1: Results for Conjunctive Queries Involving Arithmetic Inequalities

Problems	Domains	Operators	Types	Results	Reference
Satisfiability (Is S satisfiable?)	integer	$OP_{\neg\neq}$	x op y, x op c	O($\|S\|$)	[GSW96]
			x op y, x op c, x op (y+c)	O($\|S\|^3$)	[RH80]
		OP_{all}	All types	NP-hard	[RH80]
	real	$OP_{\neg\neq}$	x op y, x op c	O($\|S\|$)	[GSW96]
		OP_{all}	x op y, x op c	O($\|S\|$)	[GSW96]
			x op y, x op c, x op (y+c)	O($\|S\| + n^3$)	[GSW+96]
Implicity (Is S→T ?)	integer	$OP_{\neg\neq}$	x op y, x op c	O($\|S\|^2 + \|T\|$)	[GSW96]
			x op y, x op c, x op (y+c)	O($n^3 + \|T\|$)	[SKN89]
		OP_{all}	All types	NP-hard	[SKN89]
	real	$OP_{\neg\neq}$	x op y, x op c	O($\|S\|^2 + \|T\|$)	[GSW96]
		OP_{all}	x op y	O($\|S\|^3 + \|T\|$)	[ULL89]
			x op y, x op c	O(min($\|S\|^{2.376} + \|T\|$, $\|S\| * \|T\|$))	[GSW96]
			x op y, x op c, x op (y+c)	O($\|S\| * n^2 + \|T\|$)	[GSW+96]

disk resources are exploited, and the workload on the server side is greatly reduced. As more clients are added to the system, more resources can be used, therefore system scalability is improved. The research issues in data caching include caching granularity, cache coherency strategy, cache replacement policy, prefetching scheme, etc. In the following part, we do a survey of the previous work.

2.2.1 Caching Granularity

Caching mechanisms in traditional client-server environment are usually page-based, as they try to exploit spatial locality assuming that clustering of tuples to pages is effective. Page level caching assumes that only pages from the base relations are stored in the cache. Lots of work has been done in page-level caching, [FCL92], [FCL93], [FC94], [Fra96], [FCL97], etc.

In the work of [DM90], the caching mechanism with a granularity of a single object is implemented and compared with a page-level caching. [CSL98] investigates three different levels of granularity of caching, namely, attribute caching which caches frequently accessed attributes of database objects, object caching which caches frequently accessed objects, and hybrid caching which caches the frequently accessed attributes of those frequently accessed database objects.

The granularity of caching of [DFJ96], [KB96] and [GG97] is the result of a query, meanwhile, the semantic description is also maintained in the cache. In [Rou91], a collection of pointers pointing to the records of underlying relations needed to materialize a view is stored in the cache, i.e. the query access path is cached.

2.2.2 Caching Coherency Strategy

Cached items can become out-dated when the base data residing at the database server are updated. A cache coherency scheme is used to ensure that the cached data are consistent with those stored in the server. A cache coherency strategy can be described from the following aspects([Rou91], [BI94], [WYC96], [CTO97]):

- Update Report Content: The update report which the server sends to the clients can be
  - Data Record (Update Propagation): The update report contains the updated records.
  - Id (Update Invalidation): The update report contains only the ids of the updated records.
- Dissemination Mode: The update report can be transmitted from the server to the clients
  - Immediate (Asynchronous): Whenever there is an update, the report will be sent immediately.
  - Lazy, Deferred (Synchronous): The update report is sent periodically.
- Server State([WYC96]):
  - Stateful: The server knows which data are cached by the clients.
  - Stateless: The server is not aware of the state of the clients.
- Client's Role
  - Active: The clients query the server to verify the validity of their caches.
  - Passive: The clients listen the invalidation reports passively.

A taxonomy and summary for cache coherency schemes can be found in [BI94], [WYC96], [CTO97] and [Rou91].

2.2.3 Caching Replacement Policy

Cache replacement algorithms have been extensively studied in the context of operating system virtual memory management and data base buffer management. In order to maximize the cache hit ratio, cache replacement algorithms attempt to cache the most frequently referenced data items. The commonly used cache replacement policies include optimal, WORST, least recently used(LRU), CLOCK, etc. The optimal policy is often approximated by LRU, LRU is further generalized into LRU-k which chooses the replacement victim according to the time of the k^{th} previous access of a cached item([CSL98]).

The cache replacement algorithm in WATCHMAN([SSV96]) identifies the replacement victim by considering its average reference rate, its size and execution cost of the associated query. Instead of maximizing the cache hit ratio, WATCHMAN aims at increasing the "profit" of the cache by minimizing the execution time of queries that miss the cache.

[CSL98] develops a spectrum of replacement policies in the mobile computing environment. They base the access probabilities of database items to determine the replacement victims. For each cached item, a *replacement score* indicating the prediction of its access probability is estimated. Their first way to compute the score is by measuring the mean interoperation arrival duration. An improved approach is to use a window: the cached item with the highest mean arrival duration within the window is replaced. Their third scheme assigns weights to each arrival duration, such that the recent durations have higher weights.

In [DFJ96], the replacement strategy for semantic cache can be based on temporal locality(e.g. LRU, MRU), or on *semantic locality*. Semantic locality differs from spatial locality in that it is not dependent on the static clustering of tuples to pages; rather it dynamically adapts to the pattern of query accesses. [DFJ96] uses *semantic distance* to determine the victim, semantic regions which are closer to the most recent query are less likely to be discareded from the cache.

3 Semantic Caching Model

In this research, we assume that both the queries and the cached items are defined by *SPJ predicates*, namely, relational algebra expressions involving only projections, selections and joins. The formal definitions of the semantic cache, queries and other related concepts are given in this section. In additon, we show how to model a semantic cache.

3.1 Formal Definition

Let a database D = $\{R_i\}$ be a set of base relations. For every base relation R_i, let A_i stand for the set of its attributes. In order to define a semantic cache, we first give the definition for a set of predicates, through which the semantic cache is constructed.

Definition 1 Compare Predicate, *P, where P = a op c, a is an attribute of a base relation, op* $\in \{\leq, <, \geq, >, =, \}$, *c is a domain value or a constant.*

As shown in section 2.1, the comparison operator $\neq$ brings about complexity in reasoning among predicates, the problem even becomes NP-hard in the domain of integers. The objective of our research is to propose a novel caching scheme for client-server systems, it is not necessary to consider all the cases. Adding $\neq$ comparison would just make the predicate reasoning more complicated, and has no impact on our whole caching scheme. Hence, without a loss of generality, we ignore $\neq$ comparison in this study. Furthermore, for the join operations, we consider only the equijoin cases.

Definition 2 Join Predicate, *P, where P =* $(R_i.a_m = R_j.a_n)$, R_i, R_j *are base relations, and* a_m, a_n *are join attributes.*

Definition 3 Simple Predicate, *P, is either a Compare Predicate or a Join Predicate.*

The above assumptions and definitions provide the basis for formalizing the terminology about a semantic cache. A semantic cache consists of a set of cached items, which we define as *Semantic Segments*. Semantic segments are results of submitted queries, the definition for semantic segments is consistent with the definition of materialized views([GM95], etc.). In this study, we consider the cache items and queries that are defined only by SPJ predicates. In order to further simplify the question, so that we can concentrate on the whole cache scheme, it is assumed that the selection conditions for semantic segments and queries contain only Compare Predicates (as defined in Definition 1.

Definition 4 *Given a database* $D = \{R_i\}$*, a* **Semantic Segment** *, S, is a tuple* $< S_R, S_A, S_P, S_C >$*, where* $S_C = \pi_{S_A}\sigma_{S_P}(R_{i_1} \times R_{i_2} \times ... \times R_{i_k})$*;* $S_R = \{R_{i_1}, R_{i_2}, ..., R_{i_k}\}$*, and also* $S_R \subseteq D$*;* $S_A \subseteq A_1 \cup A_2 \cup ... \cup A_n$*;* $S_P = P_1 \vee P_2 \vee ... \vee P_m$ *where each* P_j *is a conjunctive of Simple Predicates, i.e.* $P_j = b_{j1} \wedge b_{j2} \wedge ... \wedge b_{jl}$*, each* b_{jt} *is a simple predicate.*

In Definition 4, S_P indicates the criteria which the tuples in the semantic segment S satisfy, while the actual content is represented by S_C. Furthermore, S_R and S_A define all the base relations and attributes that are involved in the semantic segmentation creation. From the restrictions added on the S_P, we can see that semantic segments are the results of Select-Project-Join (SPJ) operations, with the selection conditions containing only compare predicates.

We illustrate the definition for semantic segments with the following Example 1.

Example 1 *Consider two base relations in a database D: Employee(Eno, Ename, Age, Salary) and Project(Pno, Pname, Eno, Money). Suppose there is a query Q which will generate the semantic segment S:*
Query Q:

> *Select Ename, Pno*
> *From Employee, Project*
> *Where Employee.Eno = Project.Eno .AND. Employee.Salary > 500;*

Hence, the semantic segment S can be represented as $S = < S_R, S_A, S_P, S_C >$*, where* S_R *= {Employee, Project};* S_A *= {Ename, Pno};* S_P *= (Employee.Eno = Project.Eno)* $\wedge$ *(Employee.Salary > 500); and* S_C *contains the resulting tuples of query Q.*

To make the whole 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 on line, the cache can be regarded as consisting of a set of empty semantic segments. The formal definition will be found in Definition 5.

Definition 5 *A semantic segment, S, where* $S = < \Phi, \Phi, False, \Phi >$*, is called an* **Empty Semantic Segment***.*

We now look at the relationship between two semantic segments. Checking the relationships between semantic segments is very important in semantic caching query processing and cache maintenance which will be discussed in the following sections.

Definition 6 *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 7 *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* **Related** *iff* $S_{1_R} = S_{2_R}$.

Definition 8 *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* **Disjointed** *iff* $S_{1_C} \cap S_{2_C} = \phi$.

Three kinds of relationships between semantic segments are defined above. The *equivalent* relationship is more likely used to detect redundant data, while the *related* and *disjointed* relationships can be used in query processing, query trimming and segment coalescing, etc. Finally, the formal definition for a semantic cache is given in the following. It is defined as a set of distinct semantic segments, and is initialized as a set of empty semantic segments.

Definition 9 *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] does a very similar work as this study, however, there is no formal definition for semantic cache in [DFJ96]; furthermore, they only consider the case of selection queries on single relations. Keller and Basu in [KB96] give a formal model for *predicate-based caching*, but their model is at the cache level, they do not provide formal definition for the caching components, which we call semantic segments in our study.

Like semantic segments, we limit the queries to be only SPJ queries with equijoins and selection conditions containing only compare predicates. Hence, we have Definition 10.

Definition 10 *A* **Query** *Q is a semantic segment,* $< Q_R, Q_A, Q_P, Q_C >$.

3.2 The Semantic Cache Model

From a logical point of view, a semantic cache is composed of a set of distinct semantic segments which may vary in their sizes. However, physically, a cache is normally managed and maintained at a page level: a memory cache or disk cache is divided into pages of a fixed size. Therefore, the solution adopted for the semantic cache implementation is to page the semantic segments. Every semantic segment consists of one or more pages, all the pages are of the same fixed size, and the physical storage of the cache is managed at a page level. Notice that under this circumstance, each page contains the query results, rather that the base relations. This is different from page level caching where the cache contains pages obtained from the base relations. This idea is very similar to the *Segmentation with Paging* concept in operating system memory management([SG94]). The advantages for doing so are follows:

- *External fragmentation* can be avoided. If the physical space of the cache is maintained at a semantic segment level, as semantic segments are allocated and deallocated from the cache, the free cache space is broken into small separated pieces. External fragmentation happens when enough total free cache space exists to allocate a semantic segment, but it is not contiguous. Hence, by paging the semantic segments, the cache space can be used more efficiently.
- Allocation and deallocation algorithms are made more straightforward and simpler: For allocation, if there are enough free pages to hold the candidate semantic segment, then allocate these pages to it; for deallocation, just mark the pages which the victim semantic segment holds as free.

However, this solution has some shortcomings. As with paging, the last page of each semantic fragment generally will not be completely full. Thus, we will have, on the average, one-half page of *Internal Fragmentation* per semantic segment. Also, additional information about the relationship between semantic segments and pages should be maintained as we will discuss later.

A semantic cache is modeled to be composed of two parts: the index part and the content part. The content part consists of pages that store the tuples of semantic segments, each page has a unique page id for identification. Page numbers are physical page ids in the cache and do not relate in any way to pages in base relations. The index part keeps the following information:

- the semantic segment name S
- the set of participating base relations S_R
- the set of attributes S_A
- the predicate S_P
- the set of pages which store the semantic segment
- the timestamp indicating when the segment was last accessed S_{TS}

Table 2: Example of Semantic Cache Index

S	S_R	S_A	S_P	S_C	S_{TS}
S_1	{Stock}	{Sname}	Price < 100	{5, 8, 9}	T_0
S_2	{Company}	{Cno, Cname}	True	{6, 11}	T_0
S_3	{Stock, Company}	{Price, Location}	(Stock.Cno = Company.Cno) $\wedge$ (Business = "Computer")	{10}	T_0
S_4	{Company}	Cname	Location = "Dallas"	{7}	T_0

The model described above is consistent with the formal definition of the semantic cache. In addition to the four basic components, we add another attribute, time-stamp in the index for maintenance issues. In the following part, we explain the semantic cache structure through Example 2.

Example 2 *Consider a stock trading database with two relations: Stock(Sno, Sname, Cno, Price) and Company(Cno, Cname, Business, Location). Suppose that the semantic cache contains three semantic segments:*

- S_1:

  Select Sname
  From Stock
  Where Price < 100;

- S_2:

  Select Cno, Cname
  From Company;

- S_3:

  Select Price, Location
  From Stock, Company
  Where (Stock.Cno = Company.Cno) and (Company.Business = "Computer");

- S_4:

  Select Cname
  From Company
  Where Location = "Dallas";

Also suppose that S_1 consists of three pages, the page ids are 5, 8 and 9; S_2 consists of two pages, the page ids are 6 and 11; S_3 consists of only one page whose id is 10, and S_4 contains one page with id 7. Assume all the four segments are created at time T_0. Then the index part is shown as Table 2.

4 Semantic Caching Management

Though physically semantic segments are stored as a set of pages, the granularity of cache management is at a semantic segment level: a semantic segment is replaced and invalidated as a whole. Cache coherency control and replacement are two key components for cache management. However, the effectiveness of a cache coherency strategy is more sensitive to the application environment. For example, one cache coherency policy which is good for a client-server distributed system may not be suitable for a mobile environment. Hence, we take the cache coherency control as part of the future research. In this section, we discuss semantic caching management issues.

4.1 Query Admission

It is assumed that a semantic cache is made up of a set of semantic segments, which are results or part of results of queries submitted by the client. Initially, when the semantic cache is empty, the whole result of the first query will always be cached. However, for later coming queries, we need to decide whether to cache the query result or not, and whether to cache the whole result or only part of it. In the following apragraphs, cache admission criteria are examined. For a new query, its relationship with the existing semantic cache is illustrated with Figure 1.

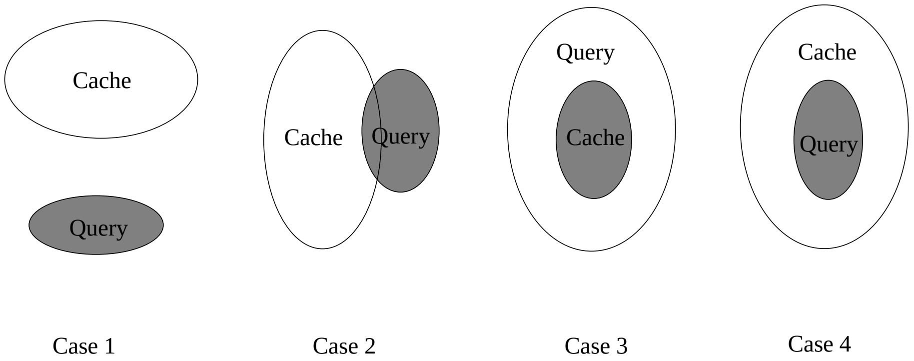


Figure 1: Possible Relations between Cache and Query

In Case 4, the query result is totally contained in the cache, there is nothing new in the query result. Therefore, this query result will not be cached again, but we need to update the maintenance information of the corresponding cached parts which are used to compute this query, such as updating the timestamps. In other cases except Case 4, we will cache part or all of the query result which is new to the cache and also update the timestamp of the old cached parts used in computing the query. Formally, the cache admission criteria can be defined as following.

Suppose a semantic cache which can be represented by a predicate C and a query Q. There are three relationship scenarios which impact whether or not query results will be added to the cache:

1. (C $\wedge Q_P$) is unsatisfiable, implying that the result of the query will not be contained in the cache. This is the same situation as Case 1 shown in Figure 1. In this case, the result of the newly coming query will be *totally admitted* to the cache.

2. $Q_P \rightarrow$ C, implying that the whole result of the query is contained in the cache. This is the same situation as Case 4 shown in Figure 1. In this case, the query result will be *ignored*.

3. Neither 1 nor 2, implying that part of the query result is contained in the cache. This is the same situation as Case 2 and Case 3 shown in Figure 1. In this case, the part of the query result which is new to the cache will be admitted to the cache. Thus it is *partially admitted*.

For practical reasons, we try to simplify the cache admission policy. Instead of comparing a query with the whole cache, we compare the query with every semantic segment in the cache and draw a conclusion. Hence, the final admission result would be the combination of results for comparing the query with several semantic segments. For example, if 1/3 result of the query is contained in segment A, the remain 2/3 is contained in segment B, then the query result will not be admitted, because it is contained in the whole semantic cache. The detail information can be found in Section 5, Query Processing.

4.2 Coalescence and Decomposition

When a query Q can be partially answered by a semantic segment S, the remainder part of the query $Q^{'}$ will be sent to the server and the answer can be obtained from there. We would then have three disjoint parts: the intersection of Q and S, the difference of S with respect to Q, and the result of $Q^{'}$, shown in Figure 2.

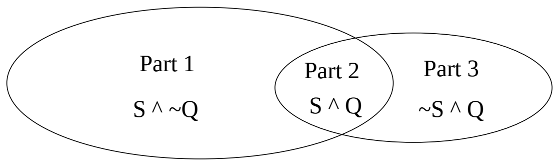


Figure 2: Relations of Three Splitted Parts

Here comes the questions: Should we always coalesce these parts to create one segment or just leave them apart? If we coalesce them, we have the following choices. Which one makes more sense?

- *Complete Coalescence:* To coalesce the three parts into one new semantic segment, i.e., to coalesce the result of remainder of Q with the original segment S.

- *Partial Coalescence:* To decompose segment S into Part 1 and Part 2, and to coalesce Part 2 and Part 3 into one segment, i.e., to make the result of query Q a new segment. This results into two semantic segments: Part 1 and Q.

- *No Coalescence:* To decompose segment S into Part 1 and Part 2, just leaving the three disjoint parts apart. This results in three semantic segments: Part 1, Part 2 and Part 3.

The advantage of *complete coalescence* is that it can reduce the number of semantic segments in the cache, thus the length of the cache index can be shortened and the cache maintenance information can be decreased. In addition, query processing is made simpler, because the number of candidate semantic segments that may contribute to the answer of a query would be reduced. However, when more and more queries come, the segment may become bigger and bigger, though only a small part of it is often used. This results in poor cache utilization. *No coalescence* can reflect the frequency of reference at a finer granularity, but results in a large number of segments. *Partial Coalescence* seems to be the best among the three approaches, it has the advantages of the other two. In this study, we will use partial coalescence.

The problem of coalescence and decomposition is relatively simpler when considering only one segment and a query. However, the result of a query may be partially contained in more than one segment, thus making the coalescence and decomposition more complicated. We will discuss the algorithm for coalescence and decomposition of the whole semantic cache and a query in the next section.

4.3 Replacement Strategy

In this study, the granularity of cache replacement is a semantic segment. The semantic caching replacement strategy is developed to be based on the *LRU(Least Recently Used)* approach. Because we do segment coalescence and decomposition as described above, both the semantic descriptions and the sizes of the segments in the cache are probably changed after a query is processed. Instead of a traditional static LRU algorithm which uses the last-visited time for the fixed pages in the cache, our replacement strategy is a dynamic one, called *D-LRU(Dynamic LRU)*, which makes use of the replacement information maintained for the dynamically changed semantic segments. We illustrate D-LRU through Example 3 (See figure 3).

Example 3 *In this example, suppose that when the query Q is issued, there are three semantic segments residing in the cache: S_1 cached at Time 1; S_2 cached at Time 2; and S_3 cached at Time 3. These segments are created after a stream of queries are processed. Also assume that Q can be partially answered by S_1 and S_2, hence, both S_1 and S_2 are decomposed into two parts separately. We keep the original cached time for the decomposed parts, S_1' and S_2', which do not contribute to the answer of query Q, and coalesce the other*

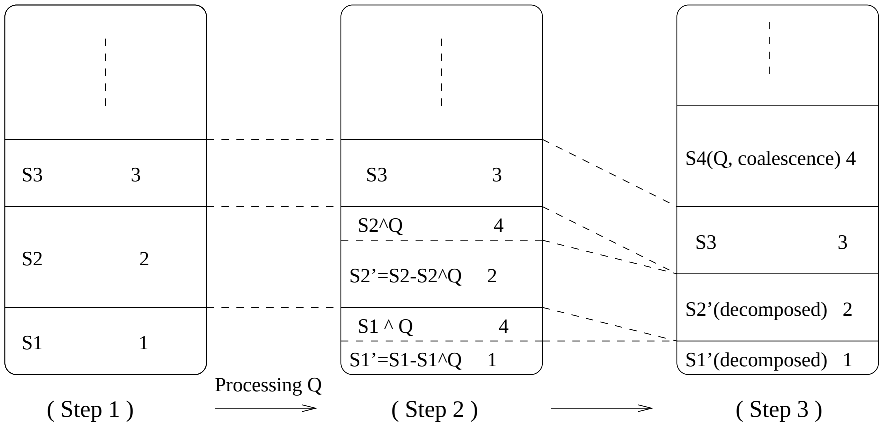


Figure 3: D-LRU through an Example

decomposed parts which contain the partial result of Q with the remainder result of Q transmitted from the server, and assign the current timestamp to it, e.g. S_4 in Figure 3.

When the cache does not have enough free space to hold the new data, the semantic segments with the oldest cached timestamp will be discarded until there is enough space. Notice that these discarded segments may not be the originally cached segments. They might be shrunk or enlarged because of decomposition and coalescence. With D-LRU, the parts of segments which are less frequently used by the queries will be separated and keep an older timestamp, hence they will eventually be replaced.

[DFJ96] proposed a replacement strategy that uses *semantic distance* to determine the victims. On one hand, extra work is needed to calculate the semantic distances; on the other hand, we believe that D-LRU can explore the advantages of the semantic distance approach, because if the segments which are "closer" to the current query are truly useful for future queries, they will be found and assigned a current timestamp by the consequent queries, and are less likely to be discarded. Therefore, in this study, we use D-LRU as the replacement strategy.

5 Query Processing

As described before, a semantic cache consists of a set of semantic segments which are defined by predicates. When a query comes, can it be partially or totally answered by the cache? If yes, how do we compute the partial or total result of the query? And if the query can only be partially answered, how do we trim the original query? In this section, techniques and algorithms for semantic caching query processing are discussed.

5.1 Formal Definition

Definition 11 *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 exists a relational algebra expression F, containing only project and select operations, such that* $F(S_C) \neq \phi$*, and* $F(S_C) \subseteq Q_C$*. If* $F(S_C) = Q_C$*, we say Q can be* **fully answered** *by S; if* $F(S_C) \subset Q_C$*, we say Q can only be* **partially answered** *by S.*

Definition 11 is derived from the idea of *Derivability* defined in [LY85]. It differs from [LY85] in two aspects. While [LY85] defines *Derivability* based on a set of derived relations, Definition 11 considers the relationship of one semantic segment and a query. In addition, Definition 11 extends [LY85] work and gives the definitions for both *fully answered* and *partially answered* cases.

Consider a semantic segment S = $< S_R, S_A, S_P, S_C >$, and a query Q = $< Q_R, Q_A, Q_P, Q_C >$. If $S_P \wedge Q_P$ is satisfiable, then S might contain a total or partial result of Q. However even all the tuples that satisfy Q are contained in S, the result of Q may not be derived from S. We explain this via Example 4.

Example 4 *For the stock trading database described in 3.1.2, consider the relation Stock(Sno, Sname, Cno, Price), suppose there is a semantic segment S and a query Q:*

$S = < \{Stock\}, \{Sname, Price\}, Price > 50, \{1, 5\} >$

Q:

select Sname
from Stock
where (Price > 100) and (Cno > 20)

It is evident that the result of Q is totally contained in S, since every tuple which satisfies (Price > 100) and (Cno > 20) will always satisfy (Price > 50). However, the result of Q can not be computed by S, because attribute Cno is needed to further qualify the tuples and Cno is not in S after the projection. Another important notice is that when $Q_A \subseteq S_A$ *does not hold, Q can't be computed from S, because some attributes of Q are not contained in S. Hence, when processing a query based on a semantic cache, we must consider not only the predicate relationship between the query and segments, but also the relationship for the attribute sets.*

Definition 12 *Consider a query* $Q = < Q_R, Q_A, Q_P, Q_C >$*, the* **predicate attribute set,** Q_{P_A}*, contains all the attributes that occur in* Q_P*, i.e.* $Q_{P_A} = \{\ a \mid a \text{ is an attribute, and } a \text{ occurs in } Q_P\ \}$

Lemma 1 *Consider a semantic segment* $S = < S_R, S_A, S_P, S_C >$*, 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 $S_R = Q_R, Q_A \subseteq S_A, Q_{P_A} \subseteq S_A$, and $Q_P \wedge S_P$ is satisfiable, then Q is answerable by S.*

2. *If $S_R = Q_R, Q_A \subseteq S_A, Q_{P_A} \subseteq S_A$, and $Q_P \rightarrow S_P$, then Q can be fully answered by S.*

Proof: Proof of 1: Suppose Q is not answerable by S, i.e., there does not exist a relational algebra expression F, containing only project and select operations, such that $F(S_C) \neq \phi$, and $F(S_C) \subseteq Q_C$.

Let us construct a query $Q^{'} =< Q_R, Q_A, Q_P \wedge S_P, Q_C^{'} >$, since every tuple which satisfies $Q_P \wedge S_P$ will always satisfy Q_P, thus we have $Q_C^{'} \subseteq Q_C$. Also $Q_P \wedge S_P$ is satisfiable, thus $Q_C^{'} \neq \phi$.

Because $S_R = Q_R, Q_A \subseteq S_A, Q_{P_A} \subseteq S_A$, and also S = $< S_R, S_A, S_P, S_C >$, then we have $Q_C^{'} = \pi_{Q_A}\sigma_{Q_P}(S_C)$. That is to say, we can find a F = $\pi_{Q_A}\sigma_{Q_P}$, such that F(S_C) = $Q_C^{'} \subseteq Q_C$ and F(S_C) $\neq \phi$. This conflicts with the assumption made at the beginning, hence Q is answerable by S. □

Proof of 2: Suppose Q can not be fully answered by S, that is to say, there exists a tuple t $\in Q_C$, and for t, there does not exist a tuple T $\in S_C$, such that t = F(T), F is a relational algebra expression.

Let us construct a semantic segment $S^{'} =< S_R, S_A, Q_P, S_C^{'} >$. Since $S_R = Q_R, Q_{P_A} \subseteq S_A, Q_P \rightarrow S_P$, we have $S_C^{'} \subseteq S_C$. Also we have $Q_A \subseteq S_A$, thus $Q_C = \pi_{Q_A}(S_C^{'})$. For tuple t $\in Q_C$, there exist at least one tuple T $\in S_C^{'}$, such that t = π_{Q_A}(T).

Because $S_C^{'} \subseteq S_C$, we have T $\in S_C$. This conflicts with the assumption made at the beginning, hence Q can be fully answered by S. □

The conditions given in Lemma 1 are sufficient conditions, there might exist less strict but more complicatedly expressed conditions that can be used to check if Q is answerable by S. The key point is to explore the relationship between attributes.

Definition 13 *Let P be a boolean expressions 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 two conditions hold:*

1. $\forall a_1, ...a_n, b_1, ...b_m, b_1^{'}, ..., b_m^{'} (P(a_1, ..., a_n, b_1, ..., b_m) \wedge P(a_1, ..., a_n, b_1^{'}, ..., b_m^{'}) \rightarrow (b_i = b_i^{'}))$

2. *For* $1 \leq i \leq n$, *there is* $\forall a_1, ...a_i, ...a_n, a_i^{'}, b_1, ...b_m, b_1^{'}, ..., b_m^{'}$ $(P(a_1, ...a_i, ...a_n, b_1, ..., b_m) \wedge P(a_1, ...a_i^{'}, ...a_n, b_1^{'}, ..., b_m^{'}) \rightarrow (a_i \neq a_i^{'}))$

Definition 13 is derived from [LY85]. However, [LY85] only gives the definition for the variables that can be *uniquely determined*. Definition 13 extends [LY85] in that it also defines the *unique* variables, i.e. the smallest set of variables that can uniquely determine other variables.

Definition 14 *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* **extendable semantic segment,** *A is the* **unique set** *of attributes of S, and $S_A \cup B$, denoted by ${S_A}^+$ is called the* **extended attribute set** *of S.*

According to Definition 14, we notice that ${S_A}^+$ is uniquely determined by A, with respect to S_P. That is to say, 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 the whole segment S to contain attributes in ${S_A}^+$. Thus, we have Definition 15.

Definition 15 *Given an extendable semantic segment $S = < S_R, S_A, S_P, S_C >$, the semantic segment $S^+ =< S_R, {S_A}^+, S_P, {S_C}^+ >$, ${S_C}^+ = \pi_{{S_A}^+}\sigma_{S_P}(R_{i_1} \times R_{i_2} \times ... \times R_{i_k})$, is called the* **extended semantic segment** *of S.*

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 Q_{P_A} is its predicate attribute set. Then we have:*

1. *If $S_R = Q_R, Q_A \subseteq {S_A}^+, Q_{P_A} \subseteq {S_A}^+$, and $Q_P \wedge S_P$ is satisfiable, then Q is answerable by S^+.*
2. *If $S_R = Q_R, Q_A \subseteq {S_A}^+, Q_{P_A} \subseteq {S_A}^+$, and $Q_P \rightarrow S_P$, 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 draw the above conclusions. □

Let us look at Example 4 again. If S is an extendable semantic segment and assume Sname is the *unique* attribute, then Cno can be *uniquely determined* by Sname. To form S^+, we can retrieve the tuples containing Sname and Cno from the database server and append them to S according to the value of Sname. After that, Q can be computed from S^+.

However, it is difficult to determine the unique set for S every time. In the following part, we give a more straightforward and convenient way to check computable problems. Assume that there is only one key attribute in every base relation, we have Lemma 3.

Definition 16 *Consider a semantic segment $S = < S_R, S_A, S_P, S_C >$, $K_A = \{a_i \mid a_i$ is the key attribute of $R_i, R_i \in S_R\}$ is called the* **key attribute set** *of S. 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* $S_R = Q_R, Q_P \wedge S_P$ *is satisfiable, then Q is answerable by* $S^+ =< S_R, S_A \cup Q_A \cup {Q_P}_A, S_P, {S^+}_C >$*.*

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

Proof: Let us at first prove that S^+ exists.

Suppose K_A is the key attribute set of S, according to K_A's definition, we know that both Q_A and ${Q_P}_A$ can be uniquely determined by K_A. S is key-contained, i.e. $K_A \subseteq S_A$, hence Q_A and ${Q_P}_A$ can be uniquely determined by S_A. That is to say, if a tuple consisting of attributes in S_A satisfies S_P, when extended to consist attributes in $S_A \cup Q_A \cup {Q_P}_A$, it will also satisfy S_P. Thus, we get S^+.

From Lemma 1, consider the relation between S^+ and Q, we can draw the above conclusions. □

Definition 17 *Consider a key-contained segment* $S = < S_R, S_A, S_P, S_C >$ *and a query* $Q = < Q_R, Q_A, Q_P, Q_C >$*,* ${Q_P}_A$ *is Q's predicate attribute set, if* $S_R = Q_R$*, then* ${S_{min}}^+ =< S_R, S_A \cup Q_A \cup {Q_P}_A, S_P, {S_{min}}^+{}_C >$ *is called the* **minimal extended segment** *of S, with respect to Q.*

5.2 Query Trimming

If a query Q can only be partially answered by a semantic segment S, it must be divided into two parts: one part that can be satisfied by S, and the other part which can not be satisfied. The part of query which can not be answered by S will be sent to the database server for processing. We call this process *Query Trimming.*

Definition 18 *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 is executed at the server to retrieve the portion of Q not found in S.*

To reduce the complexity of the query trimming, the following assumptions are made in this study. Other cases are ignored and taken as part of future research.

- Every semantic segment S = $< S_R, S_A, S_P, S_C >$ in the cache is key-contained segment.
- Only the semantic segments related to Q (see Definition 7) are chosen as the candidates to compute Q.

We categorize the relation between a query Q and a related key-contained semantic segment S into the following cases(see also Figure 4). Query trimming mechanisms for each case are also discussed.

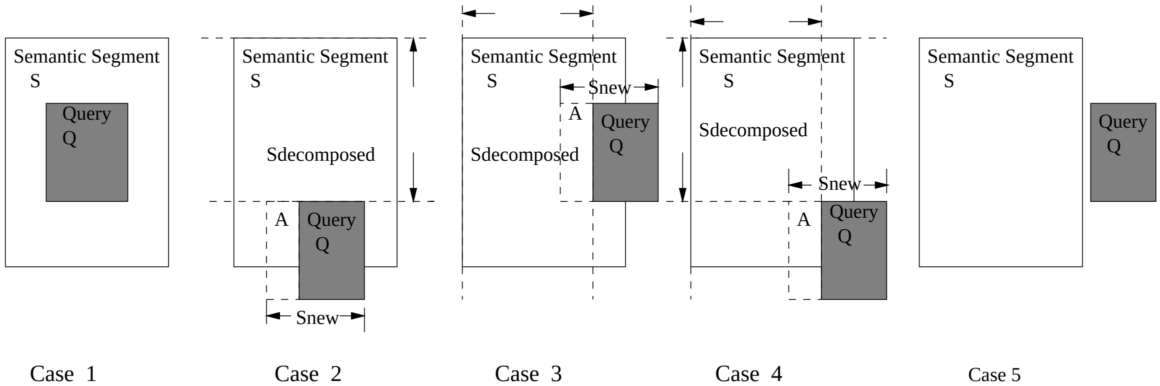


Figure 4: Possible Relations Between Q and S

Totally Contained Query Q is *totally contained* in S, i.e. $Q_P \rightarrow S_P, Q_A \subseteq S_A$, shown as Case 1 in Figure 4. Since S is key-contained, S and Q are related, according to Conclusion 2 of Lemma 3, we know that Q can be fully answered by ${S_{min}}^+$. To get ${S_{min}}^+$, we must retrieve the missing attributes from the server, we call this query *Amending Query*. Because $Q_A \subseteq S_A$, the amending query AQ for this case is:

- If ${Q_P}_A \subseteq S_A$ doesn't hold, AQ = $\pi_{({Q_P}_A \cap \neg S_A) \cup K_A} \sigma_{S_P}(R_1 \times ... \times R_n)$
- If ${Q_P}_A \subseteq S_A$ holds, AQ = NULL

After appending AQ and S according to K_A, we can get ${S_{min}}^+ =< S_R, S_A \cup Q_A \cup Q_{P_A}, S_P, {S_{min}^+}_C >$. Notice that when ${Q_P}_A \subseteq S_A$, ${S_{min}}^+$ = S. Hence, in conclusion, for the *Totally Contained* case, we have:

- If ${Q_P}_A \subseteq S_A$ doesn't hold
  - Probe Query: PQ = $\pi_{Q_A} \sigma_{Q_P}({S_{min}^+}_C)$. Processed via ${S_{min}}^+$.
  - Remainder Query: RQ = NULL;
  - Amending Query: AQ = $\pi_{({Q_P}_A \cap \neg S_A) \cup K_A} \sigma_{S_P}(R_1 \times ... \times R_n)$. Processed via base relations on server.
- If ${Q_P}_A \subseteq S_A$ holds,
  - Probe Query: PQ = $\pi_{Q_A} \sigma_{Q_P}(S_C)$. Processed via S_C.
  - Remainder Query: RQ = NULL;
  - Amending Query: AQ = NULL;

In this case, neither decomposition nor coalescence is required; no new query result is admitted to the cache; the result of Q is just the result of PQ.

Horizontally Partitioned Query Q is *horizontally partitioned* by S, i.e. $Q_A \subseteq S_A, Q_P \wedge S_P$ is satisfiable, shown as Case 2 in Figure 4. Because S is key-contained, S and Q are related, according to Conclusion 1 of Lemma 3, we know that Q is computable from ${S_{min}}^+$. To obtain ${S_{min}}^+$, an Amending Query is needed. Because $Q_A \subseteq S_A$, we have the same amending query as the *Totally Contained* case. 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 discussed in Section 3.1.3, we will do a partial coalescence after processing Q. The remainder query result will be admitted into the cache and coalesced with the part of S that contains the subset of Q. To make the resulting 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 $Q_A \cup K_A$. In conclusion, for the *Horizontally Partitioned* case, we have:

- If $Q_{P_A} \subseteq S_A$ doesn't hold
  - Probe Query: PQ = $\pi_{Q_A \cup K_A} \sigma_{Q_P}({S_{min}^+}_C)$. Processed via S_{min}^+.
  - Remainder Query: RQ = $\pi_{Q_A \cup K_A} \sigma_{Q_P \wedge \neg S_P}(R_1 \times ... \times R_n)$. Processed via base relations on server.
  - Amending Query: AQ = $\pi_{(Q_{P_A} \cap \neg S_A) \cup K_A} \sigma_{S_P}(R_1 \times ... \times R_n)$. Processed via base relations on server.
- If $Q_{P_A} \subseteq S_A$ holds,
  - Probe Query: PQ = $\pi_{Q_A \cup K_A} \sigma_{Q_P}(S_C)$. Processed via S_C.
  - Remainder Query: RQ = $\pi_{Q_A \cup K_A} \sigma_{Q_P \wedge \neg S_P}(R_1 \times ... \times R_n)$. Processed via base relations on server.
  - Amending Query: AQ = NULL;

The part of S which has been coalesced with the remainder query should be erased from S. In this case, we say S is horizontally decomposed, i.e. S's predicate is decomposed into two parts, the remainder part that doesn't contain the subset of Q is $S_P \wedge \neg Q_P$. Thus, after processing Q, decomposing S and partially coalescing, two new semantic segment are created, see Case 2 in Figure 4:

- S_{new}, the result of coalescence of PQ and RQ, $S_{new} =< S_R, Q_A \cup K_A, Q_P, S_{new_C} >$
- $S_{decomposed}$, the part of S which doesn't contain the subset of Q and is computed via deleting the tuples which contribute the answer of Q. $S_{decomposed} =< S_R, S_A, S_P \wedge \neg Q_P, S_{decomposed_C} >$.

Hence, the result of Q = $\pi_{Q_A} S_{new_C}$.

Vertically Partitioned Query Q is *vertically partitioned* by S, i.e. $Q_P \rightarrow S_P$, but $Q_A \subseteq S_A$ doesn't hold, shown as Case 3 in Figure 4. Because S is key-contained, S and Q are related, according to Conclusion 2 of Lemma 3, we know that Q can be fully answered from S_{min}^+. Since 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 attributes of Q contained in S;
- $A_2 = (Q_A \cap \neg S_A) \cup K_A$, the attributes of Q not contained in S;

To be able to compute Q via S, the missing attributes need to be retrieved from the server by an amending query and appended to S. However, this extra work can be avoided. The probe query could simply be $\pi_{A_1}(S_C)$, the tuples in the probe query result that don't satisfy Q will not appear in the final result, because the probe and remainder queries are appended according to the key attribute set K_A, and the remainder query is defined against Q. In conclusion, for the *Vertically Partitioned* case, we have:

- Probe Query: PQ = $\pi_{A_1}(S_C)$. Processed on S.
- Remainder Query: RQ = $\pi_{A_2}\sigma_{Q_P}(R_1 \times ... \times R_n)$. Processed on base relations.
- Amending Query: AQ = NULL.

The part of S which has been coalesced with the remainder query should be erased from S. In this case, we say S is vertically decomposed, i.e. S's attributes are decomposed into two parts, the remainder part that doesn't contain the attributes of Q is $S_A \cap \neg Q_A$, to make the remainder segment key-contained, K_A will be included. Thus, we have:

- Remainder Segment Query: RS = $\pi_{(S_A \cap \neg Q_A) \cup K_A}(S_C)$. Processed on S.

After processing Q, decomposing S and partially coalescing, two new semantic segment are created, see Case 3 in Figure 4:

- S_{new}, the result of coalescence of PQ and RQ, $S_{new} =< S_R, Q_A \cup K_A, Q_P, S_{new\,C} >$
- $S_{decomposed}$, the part of S which doesn't contain the attributes of Q and is computed via processing the *Remainder Segment Query RS* on S, $S_{decomposed} =< S_R, (S_A \cap \neg Q_A) \cup K_A, S_P, S_{decomposed_C} >$.

Hence, the result of Q = $\pi_{Q_A} S_{new\,C}$.

Hybridly Partitioned Query Q is *hybridly partitioned* by S, i.e. $Q_P \wedge S_P$ is satisfiable, but $Q_A \subseteq S_A$ doesn't hold, shown as Case 4 in Figure 4. Because S is key-contained, S and Q are related, according to Conclusion 2 of Lemma 3, we know that Q is answerable from ${S_{min}}^+$. We trim Q 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 horizontally partitioned case, we do a horizontal trimming on Q first. Since $Q_A \subseteq S_A$ doesn't hold, the amending query AQ is different from the horizontally partitioned case.

- Probe Query: $PQ_1 = \pi_{Q_A \cup K_A}\sigma_{Q_P}({S_{min}}^+{}_C)$. Processed on ${S_{min}}^+$.
- Remainder Query: $RQ_1 = \pi_{Q_A \cup K_A}\sigma_{Q_P \wedge \neg S_P}(R_1 \times ... \times R_n)$. Processed on base relations.
- Amending Query: AQ = $\pi_{((Q_A \cup Q_{P_A}) \cap \neg S_A) \cup K_A}\sigma_{S_P}(R_1 \times ... \times R_n)$. Processed via base relations on server.

Following the trimming mechanism for vertically partitioned case, we do a vertical trimming on PQ_1. Let $A_1 = (Q_A \cap S_A) \cup K_A$, $A_2 = (Q_A \cap \neg S_A) \cup K_A$, we have:

- Probe Query: $PQ_2 = \pi_{A_1}({S_{min}}^+{}_C)$. Processed on ${S_{min}}^+$.
- Remainder Query: $RQ_2 = \pi_{A_2}\sigma_{Q_P \wedge S_P}(R_1 \times ... \times R_n)$. Processed on base relations.

Since $A_1 \subseteq S_A$, the probe query PQ_2 can simply be $PQ_2 = \pi_{A_1}(S_C)$. In conclusion, for the *Hybridly Partitioned* case, we have:

- Probe Query: PQ = $PQ_2 = \pi_{A_1}(S_C)$. Processed on S.
- Remainder Query: RQ_1 and RQ_2. $RQ_1 = \pi_{Q_A \cup K_A}\sigma_{Q_P \wedge \neg S_P}(R_1 \times ... \times R_n)$; $RQ_2 = \pi_{A_2}\sigma_{Q_P \wedge S_P}(R_1 \times ... \times R_n)$. Processed on base relations.

The part of S which has been coalesced with the remainder query should be erased from S. In this case, we can decompose S either horizontally or vertically, the bigger one will be chosen as the final remainder segment of S, because we want to keep as much originally cached part in the cache as possible. Following the decomposing mechanisms described before, we have:

- Horizontal Remainder Segment RS_H can be obtained by deleting all the tuples in S_C which have been appended to the result of RQ_2 according to K_A.
- Vertical Remainder Segment RS_V can be obtained by processing $\pi_{(S_A \cap \neg Q_A) \cup K_A}(S_C)$ on S.

After processing Q, decomposing S and partially coalescing, two new semantic segment are created, see Case 4 in Figure 4:

- S_{new}, the result of coalescence of PQ, RQ_1 and RQ_2, $S_{new} =< S_R, Q_A \cup K_A, Q_P, S_{new_C} >$
- $S_{decomposed}$, the part of S which doesn't contain the part coalesced with the remainder query of Q. If RS_H is bigger, $S_{decomposed} =< S_R, S_A, S_P \wedge \neg Q_P, S_{decomposed_C} >$; otherwise, $S_{decomposed} =< S_R, (S_A \cap \neg Q_A) \cup K_A, S_P, S_{decomposed_C} >$.

Notice that query trimming, decomposing and coalescing can be done based on S, we don't need to get ${S_{min}}^+$. Therefore, in this case, the amending query AQ = NULL. The result of Q = $\pi_{Q_A} S_{new_C}$.

Not Contained S doesn't contain any result of Q, i.e. $Q_P \wedge S_P$ is unsatisfiable, shown as Case 5 in Figure 4. Since Q can't be answered by S, the probe query is NULL. Also because Q's result will be cached, to make the resulting segment key-contained, the key attribute set K_A should be included. In conclusion, for the *Not Contained* case, we have:

- Probe Query: PQ = NULL
- Remainder Query: RQ = $\pi_{Q_A \cup A}\sigma_{Q_P}(R_1 \times ... \times R_n)$, processed on base relations.

In this case, neither decomposition nor coalescence is required; the total Q will be cached: $S_{new} =< Q_R, Q_A \cup K_A, Q_P, S_{new_C} >$; the result of Q = $\pi_{Q_A} S_{new_C}$.

Notice that by using the query trimming mechanisms described in this section, after query processing, decomposing and partially coalescing, some originally cached parts will no longer be contained in either S_{new}

or $S_{decomposed}$(refer to figure 4). We work in this way to simplify the problem and decrease the number of new semantic segments with probably very small size.

Since both the amending query AQ and remainder query RQ are processed at the server, if the cost of amending and remainder queries are higher than the cost of processing the original query Q, we will not do query trimming. Hence:

Condition for Query Trimming: Cost(AQ) + Cost(RQ) < Cost(Q)

5.3 Query Processing

Processing a query via a single semantic segment has already been studied. While every individaul segment may contain only a small part of query result, they 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 whole cache. The algorithm for processing a query via a semantic cache is described in the following.

Algorithm 1 Res = Query_Process(C, Q), *processing a query Q via a semantic cache C*

Input: Query Q, Semantic Cache C;

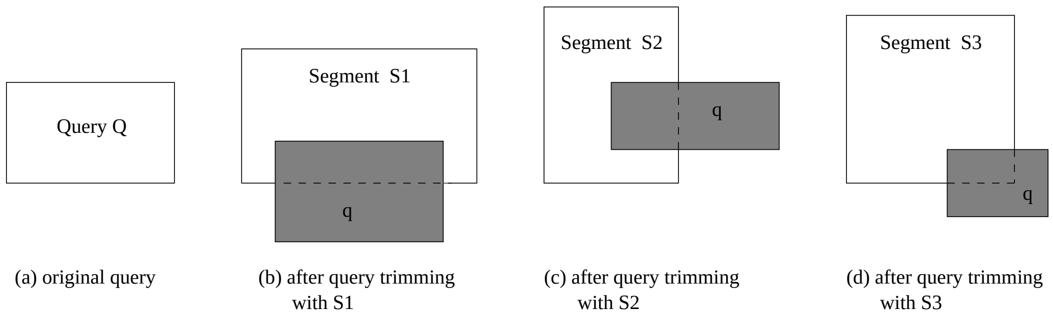
Output: Result of Q.

Procedure:

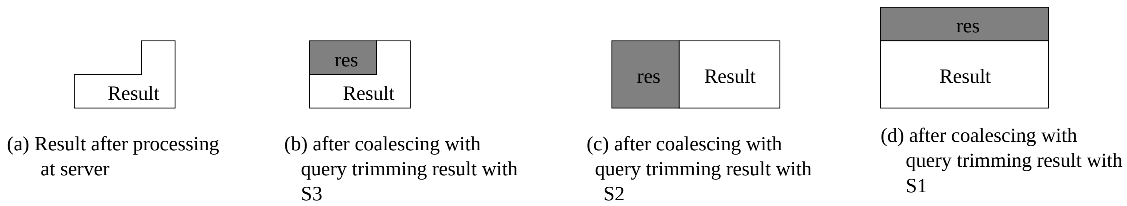
```
stack ← Empty;
T ← current timestamp;
q ← Q;
S ← the first Q related segment in C;
While ( S can be found) AND (q ≠ NULL )
{
    (Probe query PQ, Remainder query RQ) ← query_trimming(S,q);
    If ( RQ = NULL ) S_TS ← T;
    res ← PQ's result;
    push(res, stack);
    q ← RQ;
    S ← next Q related segment in C;
}
Result ← process q at the server;
while ( stack ≠ Empty )
{
    res ← pop(stack);
    Result ← coalesce(Result, res);
}
create a new segment S_new contains the result of Q;
S_newTS ← T;
If there isn't enough space, do cache replacement.
Cache S_new.
```

return(Result).
□

We explain Algorithm 1 via Example 5.



Example 5: Query Trimming



Example 5: Result Coalescing

Figure 5: Query Trimming and Result Coalescing

Example 5 *Suppose in the semantic cache, there are totally three segments, S_1, S_2 and S_3, that contain the partial result of query Q. When Q is trimmed by S_1, the probe query result is pushed into the stack for later coalescing with other part of the query; the remainder query will be trimmed by the next segment. This process continues until there is no segment that could contribute to the query result. The final remainder query will be sent to the server for processing(Refer to Figure 5). The coalescing procedure takes a reverse order from query trimming: the final remainder query result is firstly coalesced with the probe query result trimmed by S_3. This makes sense because before trimmed by S_3, they both belong to the remainder query of S_2. By poping the probe query result from the stack and coalescing with the current remainder query until there is no content in the stack, we can get the result of Q(Refer to Figure 5).*

Algorithm 1 is based on query trimming mechanisms discussed in Section 5.2. To summarize query trimming work, we give Algorithm 2 which trims a query via a related key-contained semantic segment.

Algorithm 2 (Probe Query PQ, Remainder Query RQ) = Query_Trimming(S, Q), *trimming a query Q via a semantic segment S*

Input: Query Q, Q-related key-contained semantic segment S;

Output: Probe Query PQ, Remainder Query RQ.

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_P → S_P)
{
    if (Q_A ⊆ S_A)
    {
        /***** Totally Contained *****/
        RQ ← NULL;
        If (Q_PA ⊆ S_A)
        {
            PQ ← π_{Q_A} σ_{Q_P}(S_C);
        }
        else
        {
            AQ ← π_{(Q_PA ∩ ¬S_A) ∪ K_A} σ_{S_P}(R_1 × ... × R_n);
            S_min^+_C ← append AQ and S according to K_A;
            PQ ← π_{Q_A} σ_{Q_P}(S_min^+_C);
        }
        return(PQ, RQ);
    }
    else
    {
        /***** Vertically Partitioned *****/
        PQ ← π_{A_1}(S_C);
        RQ ← π_{A_2} σ_{Q_P}(R_1 × ... × R_n);
        RS ← π_{(S_A ∩ ¬Q_A) ∪ K_A}(S_C);
        Decompose S according to RS;
        return(PQ, RQ);
    }
}
If (Q_P ∧ S_P is satisfiable )
{
    If (Q_A ⊆ S_A)
    {
        /***** Horizontally Partitioned *****/
        RQ ← π_{Q_A ∪ K_A} σ_{Q_P ∧ ¬S_P}(R_1 × ... × R_n);
        If (Q_PA ⊆ S_A)
        {
            PQ ← π_{Q_A ∪ K_A} σ_{Q_P}(S_C);
            decompose S by deleting the tuples that contribute the results of Q;
        }
        else
```

```
        {
            AQ ← π_{(Q_{P_A} ∩ ¬S_A) ∪ K_A} σ_{S_P}(R_1 × ... × R_n);
            S_min^+_C ← append AQ and S according to K_A;
            PQ ← π_{Q_A ∪ K_A} σ_{Q_P}(S_min^+_C);
            decompose S by deleting the tuples that contribute the results of Q;
        }
        return(PQ, RQ);
    }
    else
    {
        /***** Hybridly Partitioned *****/
        PQ ← π_{A_1}(S_C);
        RQ_1 ← π_{Q_A ∪ K_A} σ_{Q_P ∧ ¬S_P}(R_1 × ... × R_n);
        RQ_2 ← π_{A_2} σ_{Q_P ∧ S_P}(R_1 × ... × R_n);
        RQ ← {RQ_1, RQ_2};
        res_1 ← delete the tuples from S which can be appended to RQ_2 according to K_A;
        res_2 ← result of π_{(S_A ∩ ¬Q_A) ∪ K_A}(S_C);
        remainder part of S ← max(res_1, res_2);
        return(PQ, RQ);
    }
}
If (Q_P ∧ S_P is unsatisfiable )
{
    /***** Not Contained *****/
    PQ ← NULL;
    RQ ← π_{Q_A ∪ K_A} σ_{Q_P}(R_1 × ... × R_n);
    return(PQ, RQ);
}
□
```

Without loss of generality, we assume that the domain of the constant in the compare predicate defined in 3.1. is integer. To check the predicate relations between queries and semantic segments in the query trimming algorithm(Algorithm 2), we must define the satisfiability and implication algorithms.

Algorithm 3 Res = Disjunct_Sat(P), *to check if P, a disjunctive of conjunctives of compare predicates, is satisfiable or not.*

Input: Predicate P = $P_1 \vee P_2 \vee ... \vee P_n$, each P_i is a conjunction of compare predicates;

Output: res = Disjunct_Sat(P). If res = 1, P is satisfiable; If res = 0, P is unsatisfiable, i.e. P = False.

Procedure:

```
For ( i = 1 to n ) do
{
    /***** If P_i is satisfiable *****/
    If ( Conjunct_Sat(P_i))
```

```
            return(1);
    }
    return(0);
    □
```

Now, let us look at the algorithm Conjunct_Sat(P_i), i.e., how to check P_i, a conjunctive of compare predicates, is satisfiable or not. Let B be a conjunctive of compare predicates with variables $x_1, x_2, ..., x_k$. For each variable x_i of B, we initialize its range to $r_{x_i} = (-\infty, \infty)$, then adjust these ranges by considering each compare predicate of B in turn. Suppose before w, a compare predicate of B, is processed, the range of x_i is (a_i, b_i), r_{x_i} is then adjusted by w as follows:

- if w = $(x_i > c)$ then $r_{x_i} = (max(a_i, c), b_i)$
- if w = $(x_i < c)$ then $r_{x_i} = (a_i, min(b_i, c)$
- if w = $(x_i = c)$ then $r_{x_i} = [c, c]$

The similar rules can be applied to compare predicates with operator $\leq$ or $\geq$, we won't discuss them in detail here. Finally for each x_i, there is a range $r_{x_i} = (a_i, b_i)$ associated with it. If there exists a x_i, such that $a_i > b_i$, then B is unsatisfiable; otherwise B is satisfiable. Algorithm 4 is derived from the *INCONSISTENT* algorithm described in [LY85].

Algorithm 4 Res = Conjunct_Sat(B), *to check if B, a conjunctive of compare predicates, is satisfiable or not.*

Input: Predicate B, a conjunctive of compare predicates;

Output: res = Conjunct_Sat(B). If res = 1, B is satisfiable; If res = 0, B is unsatisfiable, i.e. B = False.

Procedure:

```
    For each compare predicate (x_i op c) in B do
    {
        find the range of x_i, r_{x_i} = (a_i, b_i);
        case op of
            <: (a_i, b_i) = (a_i, min(b_i, c));
            >: (a_i, b_i) = (max(a_i, c), b_i);
            =: (a_i, b_i) = [c, c];
        end of case;
        if (a_i > b_i) return(0);
    }
    return(1).
    □
```

From elementary logic we know that $P \rightarrow$ Q equals to $\neg(P \rightarrow Q)$ = False, meanwhile we have $\neg(P \rightarrow Q) = \neg(\neg P \vee Q) = P \wedge \neg Q$. That is to say, to check if $P \rightarrow$ Q is to check if $P \wedge \neg Q$ is unsatisfiable. It is evident that $P \wedge \neg Q$ can be expressed as a disjunction of conjunctives of compare predicates, hence Algorithm 3 can be used to check the implication problems.

6 Conclusion

In this paper, we have presented a novel caching scheme, semantic caching, and its query processing strategies. We focus on the SPJ queries involving only equijoins and with selection conditions containing only compare predicates. In addition, to reduce the complexity of semantic caching query processing, key attributes of the involved base relations are always kept in the semantic segments cached. Though with these limitations, the principles and approaches of our query processing mechanisms can still be extended to handle more complicated cases.

As far as we know, we are the first to formally define the semantic caching scheme and give the detailed semantic caching query processing techniques. We believe that with the advantages of low overhead, efficiency and flexibility, the semantic caching scheme can be used in the mobile computing environment, heterogeneous systems and general client-server environment.

For future work, we intend to apply the semantic caching scheme in the mobile computing environment. Aspects such as cache replacement strategies, cache coherency policies and prefetching methods will be explored by combining the unique characteristics of semantic caching and the mobile computing environment. The performance study for semantic caching is also a very important part of the future research.

References

[BI94] Daniel Barbara and Tomasz Imielinski, *Sleepers and Workaholics: Caching Strategies in Mobile Environments*, In SIGMOD 1994

[CFL91] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita, *Data Caching Tradeoffs in Client-Server DBMS Architecture*, SIGMOD Conference, 1991

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos and Kyuseok Shim, *Optimizing Queries with Materialized Views*, In Proceedings of ICDE, 1995

[CSL98] Boris Y. Chan, Antonio Si and Hong V. Leong, *Cache Management for Mobile Databases: Design and Evaluation*, Proceeding of ICDE, February 1998

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

[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

[DM90] David J. DeWitt and David Mater, *A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems*, Proceedings of the VLDB Conference, 1990

[FC94] Michael J. Franklin, Michael J. Carey, *Client-Server Caching Revisited*, Distributed Object Management, Morgan Kauffman, San Mateo, CA, 1994.

[FCL92] Michael J. Franklin, Michael J. Carey and Miron Livny, *Global Memory Management in Client-Server DBMS Architectures*, Proceedings of VLDB Conference, 1992

[FCL93] Michael J. Franklin, Michael J. Carey and Miron Livny, *Local Disk Caching for Client-Server Database Systems*, Proceedings of VLDB Conference, 1993

[FCL97] Michael J. Franklin, M.J. Carey and M.Livny, *Transactional Client-Server Cache Consistency: Alternatives and Performance*, ACM Transactions on Database Systems, Vol. 22, No. 3, September 1997

[Fra96] Michael J. Franklin, *Client Data Caching*, Kluwer Academic Publishers, 1996

[GG97] P. Godfrey and J. Gryz, *Semantic Query Caching in Heterogeneous Databases*, In Proceedings KRDB at VLDB'97, August 1997

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

[Gry98] Jarek Gryz, *Query Folding with Inclusion Dependencies*, In Proceedings of ICDE, 1998

[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

[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

[KB96] Arthur M.Keller and Julie Basu, *A predicate-based caching scheme for client-server database architectures*, In VLDB, 1996

[LMS95] A.Y.Levy, A.O.Mendelzon, Y.Sagiv, and D.Srivastava, *Answering Queries Using Views*, In Proceedings of the ACM Symposium on Principles of Database Systems, 1995

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

[KB96] Arthur M.Keller and Julie Basu, *A predicate-based caching scheme for client-server database architectures*, In VLDB, 1996

[Qia96] Xiaolei Qian, *Query Folding*, In Proceedings of ICDE, 1996

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

[Rou91] N.Roussopoulos, *The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis*, In ACM-TODS, Vol.16, No.3, 1991

[SG94] A.Silberschatz and P.B.Galvin, Operating System Concepts, Addison-Wesley Publishing, 1994

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

[SSV96] Peter Scheuermann, Junho Shim and Radek Vingralek, *WATCHMAN: A Data Warehouse Intelligent Cache Manager*, VLDB Conference, 1996

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

[WYC96] Kun-Lung Wu, Philip S.Yu and Ming-Syan Chen, *Energy-Efficient Caching for Wireless Mobile Computing*, In ICDE 1996

[YL87] H.Z.Yang and P.A.Larson, *Query Transformation for SPJ-queries*, In Proceedings of VLDB Conference, 1987