Efficient Query Optimization for Semantic Caching Based on the Interval Constraints Method

S. Kami Makki and Stefan Andrei

No. 4, December 2008
Efficient Query Optimization for Semantic Caching
Based on the Interval Constraints Method

S. Kami Makki and Stefan Andrei
Department of Computer Science,
Lamar University, Beaumont, Texas
kami.makki@lamar.edu, Stefan.Andrei@lamar.edu

Abstract

Semantic caching is a dynamic caching strategy which deals with not only exact but also inexact similar queries, as the local cache has the capability of analyzing the incoming queries. Therefore, in this manner, each query will be carefully analyzed by the cache manager to identify the part that can be retrieved from cache from the part that needs to be retrieved from the server. This trimming process not only speeds up information retrieval but also saves on communication cost especially for mobile devices. As, these devices have limited connection time, bandwidth, and battery power. Therefore, query trimming is the key problem in semantic caching. However, the existing methods for query trimming have a number of limitations such as, inefficiency in time, space and the complexity of the algorithm used for trimming. These factors restrict in a great extent the applicability of semantic caching for many applications. In this paper we investigate the shortcomings of query trimming process and propose a new solution to improve this process.

Keywords: Optimization, Query, Cache, semantic, trimming

1 Introduction

The recent advancement of middleware technology has provided access to many varieties of information sources. However, the network delay and lack of reliable connectivity encourage the researchers to seek solutions that can reduce the connection required to use the information sources. Commonly, different caching schemes have been employed to overcome or reduce the effect of some of the issues that exist to access these information sources, such as network connectivity, reliability and latency. For example, the results of past queries retrieved from remote information sources are cached for reuse in future queries. This can reduce the number of network accesses to these sources, and allows the sources to use their resources efficiently in serving other clients. However, traditional caching strategies such as page, tuple or attribute caching has a number of disadvantages such as the relevance of retrieved data with regards to the future queries. Since in these caching schemes, the aim is to bring relevant set of data to the cache for satisfying the future requests as the replacement strategies are generally based on the concept of locality of reference. This can be either temporal or spatial locality. The latter locality means that if an item has been referenced recently, other items that are physically close to it are more likely to be referenced. However, temporal locality means that, recently visited items are more likely to be referenced again. These specific characteristics are very well suited to the traditional data based applications (e.g., memory management) which retrieve the raw data from data sources directly without the need to use any type of query languages. However, most Internet applications are content-based applications. That is, for these applications the users need to apply the query languages for data retrieval. Therefore, these applications differ from traditional applications where the future requests for data retrieval may have similarities in the query description but the requested raw data may not stored closely in the disc. Also, the content based applications use query languages for their information gathering. Thus, in these applications the locality of reference is interpreted as semantic locality. This means, if a query has been requested earlier, other queries which have similar description (semantically related) are likely to be
requested in the near future. Thus, in these applications, the semantic locality (i.e., query similarity) is employed as the basis for cache replacement strategy, and the cache not only stores the results of the queries, but also stores the semantic description of the queries. Therefore, the cache is divided into a number of regions called semantic regions where each region contains the description and the result of a query. Then, in future retrievals, the cache manager only needs to compare the description of the new queries with description of the existing queries in the cache. If the new query or any part of it is contained in the cache, then the new query will be divided into two parts. One part is called the probe query which can be retrieved from the cache and the other part is called the remainder query which will be retrieved from the source. This trimming technique allows a good reuse of the cache contents despite of differences between the new and existing queries. This technique also can improve the performance of the cache for mobile and wireless devices because these devices are more often discounted from information sources, and the cache may provide an early result before they manage to connect to information sources. As a result, semantic caching has become very popular due to its novel method of reusing the existing information.

1.1 Related Work

The recent research in semantic caching mainly includes: research on modeling of semantic caching, query processing, maintenance strategy of consistency, and research of replacement strategy of the cache contents.

In [1], the authors defined the model and architecture of semantic caching, and provided a strategy to deal with managing the cache contents efficiently by employing efficient query processing techniques. Wu & Zhou [2] gave the sufficient condition of how the queries are able to retrieve the records from the cache using the matching relationship between the queries and the cache (exact matching, include matching, intersection matching, non-intersection matching). In [3], Wu proposed the QPID algorithm for query processing in semantic caching which is based on the disconnected environment. Their main idea is that: first, the algorithm finds the semantic region which is related to the query/queries in the cache, then the algorithm retrieves the records which are satisfying the query. Larson & Yang [4] presented the necessary and sufficient conditions for query retrieval using the materialized view, and proposed an algorithm which employs the conjunctive normal forms technique. Li et. al., [5] provided an optimized algorithm based on the historic information. But the presented method cannot be applied directly to the query processing optimization in semantic caching. Amiri & Park [6] proposed an algorithm for testing the query inclusion which is based on the model. This can promote the performance of query matching, but it does not help with the query trimming. The authors in [7] showed how to optimize the query processing for the semantic caching in two stages. However, the algorithm only is suitable for some special circumstances, so it does not provide the space-time and trimming complexity for all other cases.

Although there are many research considering query trimming from different aspect and proposing different techniques, the current strategies of query processing in semantic caching do not in-depth consider the limitations of space-time complexity of the trimming process. The existing query trimming methods have a number of limitations, such as time and space inefficiency, and complexity of the trimming process. These factors can restrict a great extent the applicability and practicability of semantic caching. Therefore, in this paper we focus on how to overcome these limitations for optimization of the query trimming process.

The paper organized as follow: In section 2 we provide the necessary definitions in regards to semantic caching. Section 3, presents an efficient query trimming algorithm and the required conditions for optimizing the query. Section 4, shows the performance of our proposed algorithm in regards to time and space complexity, and finally section 5 conclude the paper.

2 Definitions in the semantic caching

2
Semantic regions (or segments) are the basic units of storage in semantic caching. They are similar to the concept of pages in the page caching approach. However, semantic regions are dynamic in size and shape and they are based mainly on the size of retrieved data. Whereas pages in page caching strategy are static in size [7], they are not related to specificity of retrieved data. The following describes the essential definitions of a semantic caching. Given a database (DB = {R_i}, where R_i is the set of all relations). A semantic region S can be specified by the tuple <S_R, S_A, S_p, S_C>, where S_R is the set of relations in S, i.e., {R_1, R_2, ..., R_k}, and S_A is the set of all attributes of these relations, i.e., (S_A = A_1 ∪ A_2 ∪ ......... ∪ A_n in which {A_1, A_2, ...., A_n} are belong to {R_1, R_2, ..., R_k}). The relation and attributes dictate which relations and its attributes have been selected for the query. The S_p indicates the criteria that the tuples in S satisfy, and S_C represents the actual selected data or tuples in the region which satisfy the selection condition or selection predicate S_p. This can be presented, using a project operation over the result of a select operation on relations R_1, R_2, ..., R_k, which can be expressed as follow: \( S_C = \pi_{S_p} (R_1 \times R_2 \times \ldots \times R_k) \).

The selection predicate S_p can often be expressed as follow: \( S_p = P_1 \lor P_2 \lor \ldots \lor P_m \), which can make the comparative lookups much quicker and easily organized [8], where each P_j can be a conjunction of simple predicates (e.g. \( P_j = b_{j1} \land b_{j2} \land \ldots \land b_{jt} \) where each \( b_{jk} \in \{1, \ldots, t\} \) is a simple predicate, and \( b_{jk} = a \circ c \), where a is an attribute of a base relation, \( \circ \in \{\leq, <, \geq, >, =\} \), and c is a domain value or a constant). For example, the semantic segments can be illustrated as below:

Consider the following two base relations in a database DB:

- Student (StudNo, SName, Dept, CourseId, Age)
- Course (CourseId, Time, RoomNo, Instructor)

Suppose there is a query Q which will generate the semantic segment S.

**Query Q:**

\[
\text{Select StudNo, CourseId} \\
\text{From Student, Course} \\
\text{Where Student.CourseId = Course.CourseId AND Student.Dept = 'Mathematics'}
\]

The semantic segment S is represented as \( S = \langle S_R, S_A, S_p, S_C \rangle \), where \( S_R = \{ \text{Student, Course} \} \), \( S_A = \{ \text{StudNo, CourseId} \} \), \( S_p = \{ \text{Student.CourseId = Course.CourseId AND Student.Dept = 'Physics'} \} \) and \( S_C \) contains the resulting tuples of query Q or the view of the Q (when the client needs(s) the results, it will be materialized from the server).

There are also, other essential metrics of semantic caching, such as: the segment size, replacement value, a pointer to the content, and an optional timestamp. For example, the size is simply the number of tuples that are held in a region, or the number of tuples that match the condition for a particular query. Therefore, the size is not fixed and it depends on the query. The replacement value is determined by a formula that implements the cache replacement policy. A timestamp can be used for when the data in the cache was last visited or when the data was actually retrieved from the server, depending on the needs of the cache replacement policy and organizational scheme.

3 Query optimization

In semantic caching each new query is examined by the cache manager in order to identify if the query can be answered from the cache either partially or totally. If the query can be answered partially from the cache, then the cache manager can trim the original query and extract the part of query that cannot be answered from the cache. This strategy allows dividing the original query into two queries (one query retrieves the requested information from the cache contents (i.e., the probe query) and the other query retrieves the information from the server (i.e., remainder query). These two queries however, must satisfy the following conditions:
The query is a conjunctive and remainder query.

The above simple example shows that the trimming process can be constructed as follows: $Q_{\text{RQ}} = (Q_p \land S_p)$, and the predicate for the remainder query can be constructed as $Q_{\text{RQ}} = (Q_p \land \neg S_p)$ where a query can also be represented as $<Q_R, Q_A, Q_p, Q_c>$. The following simple example can demonstrate the computation of the probe and remainder queries (i.e., the query trimming process) using the above formulas for $Q_{\text{PQ}}$ and $Q_{\text{RQ}}$.

**Example 1.** Given $x$ and $y$ two integer variables, let us define $S_p$ and $Q_p$ as follow:

$S_p = (x \geq 10 \land x \leq 15 \land y \geq 5 \land y \leq 20) \lor (x \geq 5 \land x \leq 20 \land y \geq 2 \land y \leq 10)$

$Q_p = (x \geq 8 \land x \leq 17 \land y \geq 3 \land y \leq 8)$

**Probe Query:**

$Q_{\text{PQ}} = Q_p \land S_p = (x \geq 8 \land x \leq 17 \land y \geq 3 \land y \leq 8) \land ((x \geq 10 \land x \leq 15 \land y \geq 5 \land y \leq 20) \lor (x \geq 5 \land x \leq 20 \land y \geq 2 \land y \leq 10))$

**Remainder Query:**

$Q_{\text{RQ}} = Q_p \land \neg S_p = (x \geq 8 \land x \leq 17 \land y \geq 3 \land y \leq 8) \land (\neg (x \geq 10 \land x \leq 15 \land y \geq 5 \land y \leq 20) \lor (x \geq 5 \land x \leq 20 \land y \geq 2 \land y \leq 10))$

The above simple example shows that the trimming process can be very tedious, and building the probe and remainder query predicates for larger problem can be extremely complex, since the $Q_p$ of the user query is a conjunctive terms whose length is $m$, and the $S_p$ of all the semantic regions in the cache consists of the disjunction of the $n$ conjunctive terms whose length is $m$. In the worst case, the $Q_p \land S_p$ of the probe query is the disjunction of $n$ conjunctive terms whose length is $2m$, and the $Q_p \land \neg S_p$ of the remainder query is disjunction of $mn$ conjunctive terms whose length is $2n$. Therefore, both the time and space
complexity of probe query are $O(n \times m)$, while the time and space complexity of remainder query are $O(m_n)$ and $O(n \times m_n)$.

Furthermore, the remainder query predicate (i.e., $Q_P \land \neg S_P$) is a disjunction of the $m_n$ conjunctive terms whose length is $2n$ which, needs to be sent to the server. These new records will be inserted into the cache either as a new semantic region or combined with the existing semantic regions. Then the length of $S_P$ of all the semantic regions in the cache (i.e., which consists of disjunctive terms) is $m \times n + 2 \times (n \times m_n)$. As the above formula shows the number of semantic regions can increase very rapidly in a short period of time. This can cause problems for small size caches such as the type of caches used in mobile devices. These small size caches cannot provide adequate space and can fill up very quickly after a few number of queries. This reduces the applicability of semantic caching to a larger extent. Therefore, it is necessary to find a better strategy to optimize the query trimming process.

3.1 The Algorithm

The examination of query trimming process for semantic caching shows that all generated $S_P$ terms from the process of computation of probe and remainder queries contain a number of conjunctive or disjunctive terms. It also shows that, there are lots of formulas that need to be simplified. To improve this process, we employ a new and efficient technique, called the Flattening Bi-dimensional Interval Constraints (FBIC) strategy. This strategy uses semantic regions and checks efficiently the splitting and merging processes of these regions based on the queries posed.

First, we formally define the Flattening Bi-dimensional Interval Constraints problem. It is known that a bi-dimensional finite interval is given by $[a, b] \times [c, d]$, that is, $\{(x, y) \mid x \geq a \land x \leq b \land y \geq c \land x \leq d\}$. In terms of graphical image, a bi-dimensional finite interval corresponds to a finite rectangle. Therefore, the FBIC problem can be formulated as:

**Input:** a finite set of rectangles $R = \{R_1, \ldots, R_m\}$, where each rectangle $R_i$ is given by $[a_i, b_i] \times [c_i, d_i]$, $\forall i \in \{1, \ldots, m\}$;

**Output:** a finite set of rectangles $R' = \{R'_1, \ldots, R'_n\}$, such that the following properties hold:

1. $R'_1 \cup \ldots \cup R'_n = R_1 \cup \ldots \cup R_m$
2. $R'_i \cap R'_j = \emptyset$, $\forall i, j \in \{1, \ldots, n\}, i \neq j$.

The intersection operation used above in the second property of the output of the FBIC problem refers to the interior points of the rectangles. Given $R'_i = [a'_i, b'_i] \times [c'_i, d'_i]$ and $R'_j = [a'_j, b'_j] \times [c'_j, d'_j]$, then $R'_i \cap R'_j = \{(x, y) \mid x \in (\max\{a'_i, a'_j\}, \min\{b'_i, b'_j\}) \text{ and } y \in (\max\{c'_i, c'_j\}, \min\{d'_i, d'_j\})\}$. In other words, two distinct rectangles $R'_i$ and $R'_j$ such that $R'_i \cap R'_j = \emptyset$ might have in common one side, but not any interior point.

The above properties requested in the (FBIC) problem mean that each (interior) point from any rectangle $R_i$ appears as an (interior) point in exactly one rectangle $R'_j$. At the same time, each point from any rectangle $R'_i$ appears in at least one rectangle $R_i$.

As mentioned above, when a query is submitted, its semantic description will be compared with existing semantic descriptions on the cached. This allows the identification of the amount of intersection for the new query and the existing queries on the cache, and then generating the probe and remainder queries. Figures below show all possible situations of overlapping between a region and a new query. It is also possible that a query overlaps with more than one region. However, extending our approach to deal with this case is straightforward.

All regions and queries are represented as finite rectangles. As a convention, the rectangle containing the number 1 is part of the initial region, hence the other rectangle represents the new rectangle that is considered later.
Case 1: The intersection area between the two rectangles lies completely inside the second rectangle. As a result, our algorithm considers this intersecting area as being part of second rectangle.

Figure 1 - The first case of overlapping

Figure 1 shows how to merge two overlapping rectangles labeled by \{1, 2\} and \{2, 3\}, where the common area is labeled by 2. The result after overlapping is formed by two disjoint rectangles labeled by \{1\} and \{2, 3\}, respectively. In fact, the selection predicate \(S_p\) from Example 1 (Section 3) corresponds to the overlapping of the rectangles from Figure 1.

Case 2: The intersection area between the two rectangles lies completely inside the first rectangle. As a result our algorithm considers this intersecting area as being part of first rectangle.

Figure 2 - The second case of overlapping

Figure 2 shows actually a symmetric case as case 1, the only difference being that the intersection area lies inside the first rectangle, but not the second one.

Case 3: The intersection between the two rectangles gives rise to three different areas.

Figure 3 - The third case of overlapping

The new set of rectangles obtained after overlapping rectangle 3 is given by the following these three rectangles; \{1\}, \{2, 3\}, and \{5\}.

Case 4: The intersection between the two rectangles gives rise to three different areas.
The new set of rectangles obtained after overlapping rectangle 3 is given by the following these three rectangles; \{1\}, \{2, 3, 4\}, and \{5\}.

Case 5: The intersection between the two rectangles gives rise to three different areas.

Case 6: second rectangle lies completely inside first rectangle. Our algorithm consider second rectangle as a part of first rectangle.

In fact, there is one more case, in which the first rectangle lies completely inside second rectangle. However, this is the same as the case 6 which is presented above, therefore, it can be omitted.

The following shows how we can efficiently merge these rectangles. The merging operation corresponds actually with the disjunction operator and the union set operator from the rectangle definition.

### 3.2 Analysis of Algorithm

If the user query is consist of conjunctive terms whose length is \(m\), and the \(C_P\) is a disjunctive term which is made up of \(n\) conjunctive terms whose length is \(m\), then the result of negation of \(S_P\) (i.e., “NOT”
operation) is a conjunctive term made up of \( n \) disjunctive terms whose length is \( m \). This is due to the fact that \( \neg(A \land B) = \neg A \lor \neg B \) and \( \neg(A \lor B) = \neg A \land \neg B \), where “\( \equiv \)” means the equivalency relation between two propositional formulas.

In fact, we considered in Section 2 the selection predicate expressed in Disjunctive Form, that is, \( S_p = P_1 \lor P_2 \lor \ldots \lor P_m \), in which each \( P_j \) can be a conjunction of simple predicates. (Hence \( P_j \) can be denoted as \( P_j = b_{j1} \land b_{j2} \land \ldots \land b_{jn} \), where each \( b_{ji} \) is a simple predicate such as \( b_{ji} = a \circ c \), where \( a \) is an attribute of a base relation, \( \circ \in \{\leq, <, \geq, >, =\} \), and \( c \) is a domain value or a constant). Without loss of generality, that the set of inequalities \( a \circ c \) in each \( P_j \) form a finite rectangle. In fact, this apparent restriction can be easily solved by adding supplementary inequality constraints to form finite rectangles, such as, \( a \leq \text{MAXINT} \), and \( a \geq \text{MININT} \), where \( \text{MININT} \) and \( \text{MAXINT} \) represent the minimum and maximum integer values of the considered domain.

We describe now our algorithm in C programming language pseudo-code. We denote by \( \text{minX} \), \( \text{minY} \), \( \text{maxX} \), and \( \text{maxY} \) the points for newly generated rectangles. The method \( \text{fillRegion()} \) identifies the points of a region by incrementing with 0.5 the value of the \( y \) axis.

```c
fillRegion()
{
    for (minimum \( y \) to maximum \( y \), increment by 0.5) {
        for (minimum \( x \) to maximum \( x \), increment by 1) {
            (\( x, y \)) is a point in the region;
        }
    }
}
```

The method \( \text{scanRectangle()} \) scans the identified region and determines all the disjoint rectangles by incrementing with 0.5 the value of the \( y \) axis.

```c
scanRectangle()
{
    for (minimum \( y \) to maximum \( y \), increment by 0.5) {
        for (minimum \( x \) to maximum \( x \), increment by 1) {
            if (\( x == \text{minX} \)) then \( y = \text{minY} \);
                identify \( \text{maxX} \) as the last point;
            if (another point is found) {
                store new \( \text{minX} \) and \( \text{maxX} \) in array
            }
            }
        }
    }
    if (new \( \text{minX} \) and \( \text{maxX} \) are found) {
        \( y = \text{maxY} \) for a new rectangle;
    }
}
```

Obviously, the time complexity of the above algorithm for the remainder query is linear as each point of the identified rectangle is visited only once and each rectangle is corresponds with the term with 4 predicates separated by 3 conjunctive operators. Hence, the input formula of (FBIC) problem contains \( 3m \) conjunctive operators and \( m-1 \) disjunctive operators. Therefore, the time and space complexities of the algorithm are \( O(m) \), where \( m \) is the number of rectangles as defined in the input of the (FBIC) problem.

## 4 Experimental Results

We have implemented the above algorithm on a Pentium Processor with a Windows operating system which supports C programming using 2GB of RAM memory.

For example, considering the following input file with 5 rectangles:
\[\begin{align*}
x_1 \geq 1 & \land x_2 \leq 3 \land y_1 \geq 1 \land y_2 \leq 3 \\
x_1 \geq 4 & \land x_2 \leq 6 \land y_1 \geq 1 \land y_2 \leq 3 \\
x_1 \geq 2 & \land x_2 \leq 5 \land y_1 \geq 2 \land y_2 \leq 6 \\
x_1 \geq 1 & \land x_2 \leq 3 \land y_1 \geq 6 \land y_2 \leq 8 \\
x_1 \geq 4 & \land x_2 \leq 6 \land y_1 \geq 5 \land y_2 \leq 8
\end{align*}\]

our algorithm is able to find the following correct output set of 7 disjoint rectangles:

\[\begin{align*}
x_1 \geq 1 & \land x_2 \leq 3 \land y_1 \geq 1 \land y_2 \leq 2 \\
x_1 \geq 1 & \land x_2 \leq 6 \land y_1 \geq 2 \land y_2 \leq 3 \\
x_1 \geq 2 & \land x_2 \leq 5 \land y_1 \geq 3 \land y_2 \leq 5 \\
x_1 \geq 2 & \land x_2 \leq 6 \land y_1 \geq 5 \land y_2 \leq 6 \\
x_1 \geq 1 & \land x_2 \leq 3 \land y_1 \geq 6 \land y_2 \leq 8 \\
x_1 \geq 4 & \land x_2 \leq 6 \land y_1 \geq 1 \land y_2 \leq 2 \\
x_1 \geq 4 & \land x_2 \leq 6 \land y_1 \geq 6 \land y_2 \leq 8
\end{align*}\]

We have tested our program with many different test cases including various number of overlapping rectangles likely to appear in most database queries. Table 1 presents some experimental results showing the efficiency of our implementation. For example in row 3 of table 1, given 8 overlapping rectangles, our algorithm identifies 4 different and also disjoint rectangles.

<table>
<thead>
<tr>
<th>Number of Overlapping Rectangles</th>
<th>Number of different regions</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0.0031</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.0046</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0.0049</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0.0053</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>0.0055</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>0.0073</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>0.0087</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>0.0097</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>0.0114</td>
</tr>
</tbody>
</table>

Table 1 - Execution times of our algorithm

5 Conclusion

This paper discussed the necessity of the optimization for the query processing for semantic caching environment as query trimming is a key problem in semantic caching. Then it analyzed the possibility of simplifying the trimming of the probe and remainder query using FBIC.

We presented an algorithm and proved their correctness. This algorithm is used to improve the query trimming process required in the semantic caching environment. As shown, the application of FBIC reduced greatly the scale of semantic forms, which are generated during the process of simplification, and increase the efficiency of the space-time in the query trimming to have a linear complexity of the
remainder query. Hence, it makes the efficiency of the query’s processing better. Finally, all of this work offers reliable guarantee for the effectiveness and practicability of our solution.

Acknowledgment: We thank Lamar University’s students Nikunj Patel and Vishal Patel for implementing and testing the above algorithm.

6 References