A Semantic Caching Method Based on Linear Constraints

Yoshiharu Ishikawa    Hiroyuki Kitagawa
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki, 305-8573 Japan
{ishikawa,kitagawa}@is.tsukuba.ac.jp

Abstract

Because performance is a crucial issue in database systems, data caching techniques have been studied in database research field, especially in client-server databases and distributed databases. Recently, the idea of semantic caching has been proposed. The approach uses semantic information to describe cached data items so that it tries to exploit not only temporal locality but also semantic locality to improve query response time. In this paper, we propose linear constraint-based semantic caching as a new approach to semantic caching. Based on the idea of constraint databases, we describe the semantic information about the cached relational tuples as compact constraint tuples. The main focus in this paper is the representation method of cache information and the cache examination algorithm.

1. Introduction

Data caching has been investigated in various fields of database research such as client-server databases [5, 18], data warehouses [6], and distributed and heterogeneous databases [1]. In client-server environments, the method can be described as follows.

1. When a query is issued, the client cache manager checks its own cache. If part of the requested data items already exists in the cache, the client can start its local query processing.

2. To obtain the remaining part of data items, the client sends a request to the server.

3. After the client finishes its local query processing, it stores the obtained new data items into its cache. Since the cache space is bounded, it must use a cache replacement policy to decide which data items to replace when the cache is full.

As described above, we can improve query response time in client systems using caching techniques. Caching is effective when temporal locality exists in the access patterns between a query and its preceding queries; this condition holds in many cases, especially in client-server environments in which each client has its own access pattern. A caching technique, however, has an overhead to maintain cache contents to be up-to-date. Therefore, a client cache manager has a responsibility to decide which data items to be retained using a cache replacement policy.

In the traditional caching approach, cached data items (typically physical pages or tuples) were maintained based on temporal locality. Recently, some researchers have proposed the semantic caching approach, which manages cached data items using semantic locality along with temporal locality [5, 16]. In this approach, a client maintains semantic descriptions of cached data items, instead of maintaining a list of physical pages or tuple identifiers. Query processing is performed using the semantic descriptions to determine what data items are available in the local cache and what data items are to be obtained from the server. The use of semantic information encourages the effective use of cached data items.

To describe semantic information, Dar et al. used rectangular spatial regions [5]: cached data items are associated with rectangles in the semantic space. The predicate caching method, proposed by Keller and Basu, used predicates to describe cache contents [16]. In the method, predicates are used only to accumulate the reference count for each cached item and not interpreted—they are regarded as black boxes. In this paper, we extend the approach of [5] and describe semantic information using linear arithmetic constraints. We adopt the idea of constraint databases [8, 15] to represent cached tuples; the semantic information is compactly represented as a set of constraint tuples.

The rest of the paper is organized as follows. Section 2 presents a motivating example of semantic caching. In Section 3, we mention to the related work. Section 4 introduces a constraint database model to describe cached relational
tuples. Section 5 presents the cache examination and replacement algorithms. Finally, in Section 6, we give the discussion and conclude the paper.

2. A Motivating Example

Let us assume that the server contains the following relation:

\[
\text{UsedCar(name, maker, year, displacement, price, tax, ...)}
\]

In this example, we assume that the server is a relational DBMS that supports SQL queries. Suppose that a user wants to select used cars with displacement more than or equal to 1,500 cc and less than 8,000 dollars in price. The user will submit the following query to the client system:

\[
\begin{align*}
Q1: & \text{SELECT name, year} \\
& \text{FROM UsedCar} \\
& \text{WHERE displacement} \geq 1500 \\
& \text{AND price} \leq 8000
\end{align*}
\]

If the local cache is empty, the client requests the required tuples from the server using the following query:

\[
\begin{align*}
Q1': & \text{SELECT *} \\
& \text{FROM UsedCar} \\
& \text{WHERE displacement} \geq 1500 \\
& \text{AND price} \leq 8000
\end{align*}
\]

In our approach, the unit of caching is a set of *base tuples* from a server relation. Therefore, in the above example Q1’, we have modified the select clause of Q1 to ‘*’. Although this assumption introduces additional local query processing tasks, it simplifies our cache examination and replacement algorithms.

As a result of query Q1’, the client receives a set of base tuples that satisfy the query condition, and it processes the query locally. After finishing the query processing, the obtained base tuples are stored into the local cache. To describe the semantic information of the retrieved set of base tuples, we use the selection condition of Q1 as follows:

\[
q_1: (\text{displacement} \geq 1500) \land (\text{price} \leq 8000)
\]

As we will describe in Section 3, we restrict the form of a constraint to a conjunctive formula of (in)equality constraints. This semantic description is finally stored into the table *cache-info* shown in Fig. 1. The table contains other statistical information (e.g., when the base tuples are cached) for the management purpose.

Next, suppose that the user has issued the following query:

\[
\begin{align*}
Q2: & \text{SELECT name, price, tax} \\
& \text{FROM UsedCar} \\
& \text{WHERE displacement} \geq 1200 \\
& \text{AND price} + \text{tax} \leq 10000
\end{align*}
\]

The selection condition is written as a conjunctive formula as follows:

\[
q_2: (\text{displacement} \geq 1200) \land (\text{price} + \text{tax} \leq 10000)
\]

This condition is similar to \(q_1\) except for the second inequality. If we observe two selection conditions, we note that part of the cached tuples for \(q_1\) is applicable to \(q_2\); as shown in Fig. 2, two conditions overlap. The overlapped region represents the tuples that are available from the local cache. The region can be expressed by the following formula:

\[
q_1 \land q_2 = (\text{displacement} \geq 1500) \land (\text{price} \leq 8000) \\
\land (\text{price} + \text{tax} \leq 10000)
\]

Along the above explicit constraints, we assume that the client cache manager considers implicit constraints as shown below. We call such constraints *domain constraints*.

\[
\begin{align*}
d_{\text{displacement}}: & \text{displacement} > 0 \\
d_{\text{price}}: & \text{price} > 0 \\
d_{\text{tax}}: & \text{tax} \geq 0
\end{align*}
\]

A domain constraint is given as an (in)equality constraint, typically given by the database designer, and represents the range of possible values that the attribute (variable) can take.

Although we can use part of the cached base tuples to process query Q2, we still need to submit a request to obtain the remaining tuples. Such a query is called a *reminder query* and given by \(\neg q_1 \land q_2\). If the reminder query is not null (this is true for this example), it has to be sent to the server. Finally, the client updates the table *cache-info* as shown in Fig. 3.

<table>
<thead>
<tr>
<th>tid</th>
<th>constraint tuple</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>(displacement ( \geq ) 1500) \land (price ( \leq ) 8000)</td>
<td>T₁</td>
</tr>
</tbody>
</table>

![Figure 1. Cache information](image1)

![Figure 2. Overlap of constraints](image2)
3. Related Work

3.1. Semantic caching

The semantic caching approach [5, 16] uses semantic descriptions to represent the client cache contents. Dar et al. [5] used a semantic region to represent the cached data items obtained by a query. A semantic region is defined as a rectilinear spatial region and compactly represented by a simple conjunctive form such as \( q = (\text{salary} > 50000) \land (\text{age} \leq 30) \). The main difference between their approach and our proposal is that their semantic regions are restricted to rectangle shapes. Although the rectilinear region-based representation has an advantage of low calculation and maintenance costs, it cannot represent arithmetic constraints expressed by a linear combination of attributes. On the other hand, if we use the terminology of computational geometry [7], a semantic region in our context is a convex polyhedron defined by the intersection of half-spaces specified by linear arithmetic constraints. In this sense, our approach includes the rectangle region-based semantic caching approach. Linear arithmetic constraints are considered to be becoming more important because the emergence of several new database application domains (e.g., OLAP, GIS, and scientific databases) requires more efficient complex query handling facilities [9].

In the semantic region-based approach [5], a client cache manager determines whether cached data items can be used to process the given query. For its decision, it examines the semantic regions of cached items to find overlaps with the query. The best case is achieved when a semantic region or a combination of semantic regions completely covers the semantic region of the query so that the client can process the query without contacting the server. In this case, the client only issues a probe query to retrieve data items from the local cache. In other cases, the cache does not contain some or all of the required data items; the client cache manager, therefore, has to submit a reminder query to request remaining data items in addition to a probe query. Note that the query processing can be interleaved—the client query processor can start its query processing using the current cache content without waiting the completion of the reminder query. One additional advantage of the semantic region-based caching is its compact representation of reminder queries.

As another approach of semantic caching, Keller and Basu proposed the predicate caching method [16]. In this method, client cache contents are described by predicates that are contained in the preceding queries issued to the client systems. In contrast to our approach and [5], the semantics of a predicate is not used explicitly. A predicate is considered to be a black box and used to calculate reference counts for each cached data item.

We can find other approaches to semantic caching. Adali et al. proposed the intelligent cache method for distributed mediated environments [1]. It provides sophisticated use of cache mechanism using domain knowledge and additional information. Their approach was directly applied to Web caching applications by Chidlovskii et al [4].

3.2. Materialized view

The methodology of data caching has close relationship with that of materialized view management [14]. The ADMS project by Roussopoulos et al [17, 18, 19] is especially related to our research. Roughly speaking, their approach is to cache (materialize) intermediate query results and access paths, obtained while processing the incoming queries, dynamically into the local cache. The cached materialized views and access paths are reused by the subsequent queries based on the notion of subsumption, the containment relationship between queries. However, the judgment of subsumption relationship becomes complicated if we consider queries generated by arbitrary combination of relational operators. In contrast to their approach, our approach, like [5], uses a set of base tuples as a caching unit so that the judgment load is alleviated. Moreover, the semantic region-based caching uses overlap relationship between queries to decide the availability of cached items. Therefore, it has more flexibility than subsumption-based (containment-based) approaches.

3.3. Constraint databases

A constraint database is a kind of database incorporating the notion of constraints directly into the data model to model infinite information with finite representation using arithmetic constraints [3, 8, 11, 15]. The aim of constraint database model is to represent spatial or temporal information directly based on constraints. We can find many research attempts on constraint databases such as discussion of the expressive power of some constraint data model and spatial object representation scheme based on a constraint data model.

Constraints can be incorporated into database models at different levels [15]; a constraint data model is determined by specifying a base data model (e.g., relational data model...
and object-oriented data model), a database language (e.g.,
relational calculus and datalog), and a constraint domain
(e.g., dense order, linear arithmetic, and polynomials). The
constraint data model used in this paper is based on the re-
tional data model, and its constraint domain is linear arithmetic constraints over rational numbers \( \mathbb{Q} \). The reasons to use linear arithmetic constraints are: 1) it has a sufficient ex-
pressive power to represent constraints that occur in prac-
tical situations, and 2) the computational complexity to han-
dle linear arithmetic constraints is moderate one [11]. As
shown in the next section, we have not included a database
language in the constraint data model because it is sufficient
to describe our semantic cache method without the help of
a data manipulation language.

4. Data Model

In this section, we briefly define a simple constraint data
model to describe cache contents. Our model of constraint
databases is mainly based on those of Grumbach et al [11]
and Kanellakis et al. [15].

4.1. Basic definition

A constraint \( k \)-tuple (or generalized \( k \)-tuple), in variables
\( x_1, \ldots, x_k \) that range over a set \( D \), is a finite conjunction
\( \Phi_1 \land \cdots \land \Phi_N \), where each \( \Phi_i \) \( (1 \leq i \leq N) \) is a constraint. If
the arity (or the number of dimension) \( k \) is clear from the
context, we omit \( k \), and use the term constraint tuple. There
are a lot of kinds of constraint tuples depending on the kinds
of constraints used. In all case, equality constraints and
inequality constraints among individual variables and con-
stants are allowed. For example, \( (x = 3 \land y \neq 2) \) represents
a constraint 2-tuple. We can observe that a relational tuple
(e.g., \( (3, 2) \)) can be expressed by using equality constraints
as above. Therefore, we can interpret any relational tuples
as constraint tuples.

A constraint relation of arity \( k \) (a generalized relation of
arity \( k \)) over a set \( D \) is a finite set \( r = \{ \varphi_1, \ldots, \varphi_M \} \), where
each \( \varphi_i \) \( (1 \leq i \leq M) \) is a constraint \( k \)-tuple in the same vari-
ables \( x_1, \ldots, x_k \). The formula corresponding to a constraint
relation \( r \) is the disjunction \( \varphi_1 \lor \cdots \lor \varphi_M \). It is in dis-
junctive normal form (DNF) of constraints, which uses at
most \( k \) variables ranging over set \( D \). Each constraint re-
lation describes a finite set of \( k \) tuples (or points in
k-dimensional space \( D^k \)) and represents a possibly infinite
set of \( k \) tuples. A constraint database is a finite set of
constraint relations.

4.2. Linear constraints

A constraint in a constraint data model is usually repre-
sented by a first-order language and interpreted over some
numeric domain (e.g., rational numbers \( \mathbb{Q} \) and real numbers \( \mathbb{R} \)). The constraints we consider in this paper belong to the
first-order language \( \mathcal{L} = \{ \leq, + \} \cup \mathbb{Q} \). This class of language
has a good tradeoff between the query processing power
and its computational complexity [10, 13]. In addition to
the term constraint domain, we use the term uninterpreted
domain to specify an attribute domain that cannot be asso-
ciated to numeric values. We assume that only constraints
regarding ‘=’ and ‘≠’ can be specified over an uninterpreted
domain.

A constraint over rational numbers \( \mathbb{Q} \) has the following
form:

\[
ax_1 + \cdots + a_p x_p \theta a_0,
\]

where \( x_i \) is a variable, \( a_i \) is an integer constant \( (1 \leq i \leq p) \),
and \( \theta \) satisfies

\[
\theta \in \{ =, \neq, <, \leq, >, \geq \}.
\]

The term \( ax_i \) is an abbreviation of \( x_i + \cdots + x_i \) \( (a_i \) times).

In this paper, we use linear arithmetic constraints along
with the constraint data model defined in subsection 4.1 as
the data model to describe the semantic information in a
client cache. A constraint relation defined as above some-
times called a linear constraint relation.

4.3. Orthographic partitioning

The computational complexity of processing constraints
highly depends on the class of constraint domain and the
number of variables (i.e., the number of dimensions) con-
tained in constraints. It is problematic especially for large
databases that contain many attributes. To cope with this
problem, Grumbach et al put restrictions on linear con-
straint relations and gave an upper bound of the query com-
plexity [12]. Although their context is quite different from
ours (since we do not need the full expressive power of the
linear constraint data model), their idea can be applicable to
reduce the processing cost in our context.

The idea is based on the idea of dependent variables: if
two variables appear in the same linear constraint, we
call these variables dependent; otherwise they are inde-
pendent. For example, in the example in Section 2, vari-
ables (attributes) price and tax are dependent because they
have appeared in the same constraint. On the other hand,
displacement and price are independent because they usu-
ally do not appear in the same constraint formula. Based
on this idea, we can group variables into some orthographic partitions in terms of dependency. In the exam-
ple of Section 2, we can obtain two orthographic partitions:
\( \text{op}_1 = \{ \text{displacement} \} \) and \( \text{op}_2 = \{ \text{price}, \text{tax} \} \). If we incor-
porate the notions of dependency and orthographic parti-
tion, the cost of constraint manipulation does not depend on
the total number of variables—it depends on the partition
with the largest number of variables. The number of variables of the largest partition is called orthographic dimension. Orthographic dimension is an important factor to estimate upper bounds of operations over constraint databases.

If we incorporate the idea of orthographic partition, we can represent cache-info relation shown in Fig. 3 as in Fig. 4. Namely, each orthographic partition is stored independently in cache-info relation. This representation will reduce the computation cost and introduce flexibility to incorporate indexes for the efficient query processing (see Section 7).

![Figure 4. Partitioned representation](image)

5. Using Cache Information

In this section, we show how to use the cache information represented by a constraint relation to generate the probe query and the reminder query for a given query.

5.1. Cache examination algorithm

As described in Section 2, when a client receives a query \( q \), it examines the constraint tuples in the cache-info relation to generate the probe query \( pq \) and the reminder query \( rq \). Figure 5 shows the algorithm for the examination of cache information. The algorithm is naive in the sense that it examines all the constraint tuples in cache-info; we will present some ideas to improve the algorithm in Section 7.

We have to make some definitions to explain the algorithm. Let the number of constraint tuples in cache-info relation be \( n \) and the number of orthographic partitions be \( k \). Then assume that the given conjunctive query has a general form \( q = q_1 \land q_2 \land \cdots \land q_l \), where each \( q_i \) corresponds to \( i \)th orthographic partition \( op_i \). If \( q \) has no corresponding condition for \( op_i \), let \( q_i = T \). The symbol \( 'T' \) means the Boolean truth value; the false value is defined as \( F = \neg T \). For example, the constraint

\[
\text{(displacement} \geq 1500) \land \text{(price} \leq 8000),
\]

appeared in Section 2, has \( q_1 = \text{(displacement} \geq 1500) \) and \( q_2 = \text{(price} \leq 8000) \) for the orthographic partitions \( op_1 = \{\text{displacement}\} \) and \( op_2 = \{\text{price}, \text{tax}\} \), and the constraint

\[
\text{(price} \leq 8000) \land \text{(price} + \text{tax} \leq 10000)\]

has \( q_1 = T \) and \( q_2 = (\text{price} \leq 8000) \land (\text{price} + \text{tax} \leq 10000) \). In the algorithm, we assume that \( q_1 \neq T \) for \( i = 1, \ldots, l \) and \( q_i = T \) for \( i = l + 1, \ldots, k \) to simplify the presentation.

![Figure 5. Naive cache examination algorithm](image)
5.2. Local query processing

When a query $q$ is given, the client begins its query processing. The client query processing steps are shown below.

1. The client calculates the probe query $pq$ and the reminder query $rq$ based on the cache examination algorithm.

2. If $pq \neq F$, the client evaluates $pq$ to retrieve tuples from its local cache.

3. If $rq \neq F$, the client sends $rq$ to the server to obtain the remaining tuples then continues its query processing.

4. After the query processing is finished, the client stores all the obtained tuples into its local cache (if the cache area is exhausted, the client triggers the cache replacement procedure—see Section 7).

5. Finally, the client adds a new tuple into cache-info and updates other statistical information. For example, it increments the reference counter for each tuple accessed (and cached) by the query $q$.

In the above algorithm, we have not considered domain constraints to simplify the presentation. It is easy to incorporate domain constraints into the above algorithm.

### 5.3. Overlap computation

In subsection 5.1, we did not mention how to calculate compute-overlap. In this subsection, we explain its calculation method. Before we describe the method, two requirements for compute-overlap are shown:

1. It calculates the overlap of two given conjunctive constraints: if the overlap is empty, the function returns the false value $F$.

2. It is desirable to delete redundant constraints from the given constraints. For example, it must reduce a constraint $(age > 30) \land (age \leq 50) \land (age > 40)$ to $(age > 40) \land (age \leq 50)$. For simple constraints (such as 1-D or 2-D rectilinear constraints), we can easily detect redundant constraints. However, it is not obvious to detect redundant constraints for linear arithmetic constraints.

To calculate the overlap region of two constraints, and to detect redundant constraints, we extend the idea found in linear programming literature [20]. Note that the procedure given below is rather general one; we can choose algorithms that are more efficient by observing specific conditions satisfied in each situation. For example, if the number of dimension is one or two, we can use more efficient computation methods using the techniques found in computational geometry [7].

First, we normalize the given constraints:

1. Decompose the constraints into constraints in the normal form (Eq. (1)). For example, $0 < x \leq 100$ is transformed into $x > 0$ and $x \leq 100$.

2. If a constraint has the form $\alpha x_1 + \cdots + \alpha p x_p = \alpha_0$, transform it, for example, into the form $x_1 = \frac{\alpha_0}{\alpha_1} (\cdots)$, for example, and then delete $x_1$ from all of the constraints. Namely, we can reduce the number of variables (dimensions).

3. If the constraint has the form $\alpha_1 x_1 + \cdots + \alpha_p x_p \geq \alpha_0$, rewrite it to $\alpha_1 x_1 + \cdots + \alpha_p x_p > \alpha_0$, the same transformation is applied to the case of ‘$<$’.

After step 3, all the constraints have the following form: $\alpha_1 x_1 + \cdots + \alpha_p x_p \theta \alpha_0$ ($\theta \in \{\leq, \geq\}$). In the following steps 4–7, we first process constraints with the form:

$$
\alpha_1 x_1 + \cdots + \alpha_p x_p \theta \alpha_0, \quad \theta \in \{\leq, \geq\}.
$$

We consider constraints according to ‘$\neq$’ after the main process has finished.

The problem of calculating the overlap of linear constraints with the form (3) and the problem of selecting non-redundant constraints that exactly define the overlap can be
reduced to the problem of calculating all the basic feasible solutions from the given linear constraints [20]. We briefly describe the procedure:

4. By incorporating slack variables [20] and replacing variables appropriately, we can transform all the constraints into the following form \((n > m)\):

\[
\begin{align*}
    a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + \cdots + a_{2n}x_n &= b_2 \\
    \vdots & \quad \vdots \\
    a_{m1}x_1 + \cdots + a_{mn}x_n &= b_m
\end{align*}
\]

If the overlap exists, solutions from the given linear constraints \([20]\). We briefly reduced to the problem of calculating all the basic feasible solutions. This is achieved by solving \(Ax = b\) and replacing the additional constraint \(a \neq 0\) does not interfere with \(c\). Otherwise, replace \(c\) with \(c' = c \land (a_{11}x_1 + \cdots + a_{n}x_n \neq a_0)\).

9. We have to compensate the effect of step 3 where 'greater than' and 'less than' are processed as 'greater than or equal to', respectively. As an example, consider that a constraint \(c_a = a_{11}x_1 + \cdots + a_{n}x_n \geq a_0\) was replaced into \(c_b = a_{11}x_1 + \cdots + a_{n}x_n \geq a_0\) in step 3. If we can find \(c_b\) in the set of non-redundant constraints \(\{c_1, \ldots, c_r\}\) obtained in step 7, replace \(c_b\) with the original constraint \(c_a\). If we cannot find \(c_b\) in \(\{c_1, \ldots, c_r\}\), do nothing.

If the overlap exists, \(\text{compute-overlap}\) returns the final conjunctive constraint \(c = c_1 \land \cdots \land c_r\) as a result.

6. Cache Replacement Algorithm

Since the cache area in the client system is bounded, we usually need to replace old cached items with incoming new data items. For the cache replacement problem, we must clarify 1) the selection method of a victim (the target of replacement), and 2) the cache replacement algorithm.

To select a victim, the client cache manager has to consider several factors such as the freshness and the access frequency of the cached items and uses a cache replacement policy (e.g., LRU) to make a decision. It depends on the situation which kind of cache management mechanism is appropriate. Therefore, here we do not specify the actual cache management method and simply assume that the \(\text{select-victim}\) function, which selects an appropriate victim, is available.

The cache replacement algorithm is shown in Fig. 7. We denote the constraint tuple specified as a victim by \(tv\). In the algorithm, a simple reference count-based method is used. For the selected victim (constraint tuple) \(tv\), the corresponding base tuples \(V = \{bt_1, \ldots, bt_n\}\) are retrieved from the cache and then the reference count for each \(bt_i\) is decremented by 1. Finally, the cache manager deletes all of the base tuples with reference count 0.

1. \(tv := \text{select-victim}()\);
2. let \(q_v\) be the constraint corresponding to \(tv\);
3. submit query \(q_v\) to the local cache;
4. let \(V = \{bt_1, \ldots, bt_n\}\) be the query result;
5. \(\text{for } i := 1 \text{ to } n \text{ do } \{
\text{6. } \text{ref.count}(bt_i) := \text{ref.count}(bt_i) - 1;
\text{7. } \text{if } (\text{ref.count}(bt_i) = 0) \text{ then }
\text{8. } \text{delete } bt_i \text{ from the cache};
\text{9. }
\text{10. } \text{delete } tv \text{ from cache-info}.
\}\)

Figure 7. Cache replacement algorithm

To make this replacement algorithm work correctly, the cache manager has to maintain the reference count for each cached base tuple. Therefore, when a base tuple is accessed by a probe query or a reminder query, the cache manager has to increment its reference count correctly.

7. Discussion and Conclusion

The cache examination algorithm described in subsection 5.2 scans all the constraint tuples in \(\text{cache-info}\) relation. This causes a critical problem when \(\text{cache-info}\) relation is large. One of the solutions to this problem is the use of indexes. For constraint databases, there have been several proposals of indexing techniques [2]. Since indexing methods for constraint databases depend on the number of variables (dimensions), the actual constraint maintenance
cost depends on orthographic partitions that determine the actual number of dimensions to be considered. Therefore, the selection method of indexes also depends on them.

If the number of variables (dimensions) of an orthographic partition is one or two, efficient indexing methods for constraint databases are available [2]. If the number of dimensions is more than two, two approaches exists: 1) Project each constraint on one or two dimensions, and then use above-mentioned indexing methods, or 2) Extract minimum bounding boxes (MBBs) from each constraint then use them as spatial indexes. Such a spatial index can be used in the filtering step to select the candidates of constraint tuples that may satisfy the condition.

To reduce the computational overhead of the naive algorithm shown in subsection 5.1, we can also use indexes and approximations. When a query \( q \) is given, we first search the candidate constraint tuples using indexes. Next, we can use approximation-based comparisons (e.g., using MBBs to approximate constraints) to filter out unnecessary constraint tuples. Then we start the exact comparison by the algorithm shown in this paper.

In this paper, we have proposed a new semantic caching approach based on linear arithmetic constraints. We adopted the idea of constraint databases to represent cached base tuples by compact constraint tuples. This paper mainly focused on the cache examination algorithm to utilize cached information, the method to calculate the overlap of two semantic regions, and the cache replacement algorithm. The remaining issues are the cache examination algorithm incorporating approximation, the use of indexes for the efficient cache management, and development of efficient cache replacement policies and their comparisons.

Acknowledgments

This work was partially supported by the Grant-in-Aid for Scientific Research from the Ministry of Education, Science, Sports and Culture of Japan.

References