A Boolean Query Processing with a Result Cache in Mediator Systems

Jae-heon Cheong and Sang-goo Lee*
Department of Computer Science
Seoul National University
San 56-1 Shillim-dong Kwanak-gu, Seoul Korea
{cjh, sglee}@cygnus.snu.ac.kr

Abstract

A mediator system is a kind of a meta-search engine that provides a seamlessly integrated search service for diverse search engines (collections). Since collections of a mediator system are geographically distributed, its performance is mainly influenced by the data transmission time between the mediator and its collections. Existing mediator systems employ a result cache that is composed of the results of previously issued queries to reduce this transmission time. However, these systems do not support a general Boolean query model but only simple ones such as a conjunctive query, a single keyword query, and or so. In this paper, we propose a new method to efficiently process general Boolean queries using a result cache for mediator systems. Also presented is a way to semantically partition the given result cache in order to reduce the complexity of inference.

1. Introduction

The huge number of information sources and search engines on the Internet makes integrated search services, or meta-search services, almost inevitable. The mediation architecture [1] is a natural choice for a meta-search system, in which a mediator provides users with seamless access to information from distributed and heterogeneous resources.

Since the information sources and search engines that a mediator system deals with are geographically distributed over the network, the data transmission time between the mediator and its sources is a determining factor in overall performance. In order to reduce the size of the query results that have to be transferred, most mediators make use of the results of previous queries, typically stored in a result cache. Current methods of query processing that utilize the result cache support only simple query models that do not allow general Boolean queries.

*Author’s work was supported in part by research fund provided by the Korea Research Foundation, Support for Faculty Research Abroad

Cache representation and inference based on the general Boolean model gives rise to complicated problems. First of all, it is not practical for a result cache to be equipped with a keyword index, such as an inverted file, since it cannot be known in advance what keywords will occur in a query. Thus, if a query contains at least one keyword not in the result cache, the entire cache must be searched exhaustively for proper sub-results that satisfy the current query. Another complication arises when information sources do not support the ‘NOT’ operator. To retrieve the portion of the result for a query \( Q \) that is not already in the result cache \( C \) (we call this a miss result), a new query ‘\( Q \) AND NOT \( C \)’ (the miss query) would be sent to the sources. If, however, some of the sources do not support the ‘NOT’ operator, the mediator cannot retrieve those results directly.

We propose a new method to solve these problems and to efficiently process general Boolean queries using a result cache. We present a method that utilizes the result cache to minimize the result size that must be retrieved from the sources for a keyword-based general Boolean query. The miss results can be computed effectively even when the target collections do not support the ‘NOT’ operator. Also presented is a way to semantically partition the given result cache in order to reduce the complexity of inference.

In the next section, we present other works related to the current topic. A basic cache-based query-processing model is presented in section 3. In section 4, we propose our base method of cache-based query processing. In section 5, we present a method to semantically partition the current result cache and how to process general Boolean queries with a partitioned result cache. Performance issues are considered in section 6. We conclude the paper in section 7 with a summary of our contributions and a brief discussion on future directions.

2. Related works

There have been a number of articles on semantic data caching [2, 3, 4, 5, 6]. In [2], the detailed method of rewriting an initial query using a cached query is
described. However a single database environment is assumed and the algorithm for finding a query match (a hit query in our context) is incomplete. A query optimization method is proposed in [3], which uses cached queries in a mediator system named HERMES. This system, however, deals with only simple conditions of the form $c_1 \theta c_2$ (where $\theta$ is any of $<$, $\leq$, or $=$). Cache replacement strategies based on recency and semantic distance of cached queries are considered in [4], while consistency strategies based on recency and semantic distance of cached queries are considered in [4]. The query approximation method presented in [6] also considers only conjunctive queries.

Perhaps the work most closely related to semantic caching in the database literature is [7], which deals with materialized views. This work, however, deals only with conjunctive queries in SQL. A scheme for rewriter queries under the Boolean query model considering the capabilities of information sources is presented in [8]. The rewriter is based on the exact-predicate matching.

In [9], a query history based virtual index (QVI) is proposed. It is based on the observation that queries are repeated over time and users. However, only simple keyword queries like ‘Retrieve documents that contain keyword 1 and keyword 2’ were considered. We propose to extend this basic idea to a keyword-based general Boolean retrieval model.

3. Preliminaries

In this section, we present some basic concepts necessary to build our new method. We adopt the keyword-based Boolean query model, where a query, for instance, “Retrieve documents that contain both keywords ‘digital’ and ‘library’” can be represented by a Boolean expression ‘digital AND library’.

For notational convenience, we use the symbol ‘$\land$’, ‘$\lor$’, and ‘$\neg$’ to denote the Boolean operators ‘AND’, ‘OR’, and ‘NOT’, respectively.

3.1. Query model

In a keyword-based Boolean query model, queries are defined recursively as follow:

**Definition 1: Query**
1. A keyword in a query.
2. If $A$ is a query, then ($\neg A$) is also a query.
3. If $A$ and $B$ are queries, then ($A \land B$) and ($A \lor B$) are also queries.
4. All queries are generated by applying the above rules.

On the Internet, the target of a query is the set of documents. A document can be defined as a conjunction of keywords that occur in it. For example, a document that contains keywords such as ‘database’, ‘information retrieval’, and ‘mediator’ can be represented by ‘database $\land$ information retrieval $\land$ mediator’.

For a given query $Q$, the query result of $Q$, denoted by $[Q]$, is defined as the set of documents that satisfy $Q$. In general, a result of a query varies as a database changes. However, the change of a document database’s state is not dominated by updating or deleting existing documents but by appending new documents in the information retrieval environment on the Internet. So, we can assume that the state of a document database is static without loss of generality.

**Example 1:** Suppose a set of documents is $\{t_1, t_2 \land t_3, t_1 \land t_5 \land t_6, t_4 \land t_6, t_5 \land t_6\}$. Then, the result of a query $(t_1 \lor t_2) \land (t_3 \lor t_6)$ is $\{t_2 \land t_3, t_1 \land t_5 \land t_6\}$.

If two different queries have the same result, we say that these two queries are equivalent. A query is a sub-query of another query if the result of the former is fully contained in that of the latter. To simplify our query model, we assume that all keywords are independent of each other.

3.2. Cache-based query processing model

A result cache is defined as the collection of previously issued queries and their results. Since queries are represented by Boolean expressions, a result cache is represented by the disjunction of previously posed queries. The contents of a result cache are the union of previous results.

**Definition 2: Result Cache**
Suppose $Q_1, Q_2, \ldots, Q_n$ are queries that have been issued by users up to this point. The current result cache, denoted by $C$, is defined as the disjunction of previous queries $(Q_1 \lor Q_2 \lor \ldots \lor Q_n)$.

**Example 2:** Suppose queries that have been issued until now are $(t_1 \lor t_2), (t_3 \lor t_4), (t_2 \land t_3)$, and $(t_1 \land t_2 \land t_4)$, then the current result cache is represented by $(t_1 \lor t_2) \lor (t_3 \lor t_4) \lor (t_2 \land t_3) \lor (t_1 \land t_2 \land t_4)$.

When a new query is submitted, it can be decomposed into two sub-queries such as a hit query and a miss query. While a hit query can be answered from the current result cache, a miss query must be transferred to the appropriate collections to get answers.

**Definition 3: Hit Query (HQ)**
For a given query $Q$, if a query $Q_C$ subsumes $(Q \land C)$, then $Q_C$ is called a hit query for $Q$ and $C$. $Q_C$ is called the optimal hit query ($OHQ$) if $Q_C$ is equivalent to $Q \land C$. 
Definition 4: Miss Query (MQ)
For a given query \( Q \), if a query \( Q \land C \) subsumes \( Q \land \neg C \), \( Q \land C \) is called the miss query for \( Q \) and \( C \). \( Q \land C \) is called the optimal miss query (OMQ) if \( Q \land C \) is equivalent to \( Q \land \neg C \).

If the result of \( Q \land C \) is not empty and \( Q \) is not equivalent to \( C \), then the original query \( Q \) can be both a hit query and a miss query for \( Q \) and \( C \) by itself. If \( Q \) is equivalent to \( C \), the optimal hit query for \( Q \) and \( C \) is equivalent to \( Q \). And if \( Q \land C \) is empty, the optimal miss query for \( Q \) and \( C \) is equivalent to \( Q \).

Example 3: Suppose a query \( Q \) is \( t_2 \land (t_1 \lor t_3) \) and the current cache \( C \) is \( t_1 \lor (t_2 \land t_3) \). Then, the OHQ is \( Q \land C = (t_1 \land t_2) \lor (t_1 \land t_3) \lor (t_2 \land t_3) \) and the OMQ is \( Q \land \neg C = (t_2 \land \neg (t_1 \lor t_3)) \).

The procedure to process queries using a result cache is composed of 3 steps.

Step 1: Query Decomposition
When a query is posed, the mediator decomposes the query into a HQ and a MQ. In general, the initial query can be used as both HQ and MQ.

Step 2: Retrieve Results
The mediator retrieves cached results from the cache by applying HQ, and un-cached results by sending MQ to the appropriate search engines. If we use an OHQ, we can reduce the time to retrieve data from the cache. Using an OMQ can minimize the size of the results that should be transferred from the search engines.

Step 3: Integration of Results & Cache Update
The final result can be generated by integrating the results from the cache and those from the data sources. The cache is updated with the results retrieved from the data sources.

3.3. Problem statement
We should have reasonable and practical solutions in order to realize the above procedure. First of all, if we want to minimize the whole computation time, we should decompose the initial query into an OHQ and an OMQ. Unfortunately, however, it might be costly if the decomposition process employs a general logic inference. In addition, splitting up a query into two parts leads to more complicated queries that might be too expensive to process. Second, if the size of a cache is large, the time required to get answers from the cache can be quite long. Third, since it is common that miss queries contain one or more ‘NOT’ operators and there might exist some search engines that cannot directly answer those miss queries because they do not support the ‘NOT’ operator. To those search engines, the mediator should send a query that subsumes the miss query and does not contain any ‘NOT’ operator. Unfortunately, however, it is expensive and complicated to compute those subsuming queries.

4. Cache-based Query Processing
We propose a new description method based on the propositional logic for queries and a result cache in this section. We also propose algorithms to process queries with a result cache.

4.1. Unfolded Disjunctive Normal Form (UDNF)
We define Unfolded Disjunctive Normal Form (UDNF) to be a variation of the DNF where every conjunct is pair wise disjoint or unsatisfiable. This means that the intersection of the result sets of any two conjuncts is always empty.

Definition 5: UDNF
A formula \( F \) is said to be in an unfolded disjunctive normal form if and only if \( F \) has the form of \( F = F_1 \lor F_2 \lor \ldots \lor F_n \), where every \( F_1, \ldots, F_n \) is pair wise disjoint.

If a result cache is transformed to its UDNF, its result is partitioned into several disjoint sub-results represented by conjuncts respectively. There exist many ways to transform a Boolean expression to its UDNF. In this paper, we represent UDNF by means of minterm. A minterm for a given set of keywords is defined as a conjunction in which every keyword occurs exactly once, either in its positive or negative form [10]. For instance, if the set of keywords is \( \{t_1, t_2, t_3\} \) then the set of minterms is \( \{t_1 \land t_2 \land t_3, t_1 \land t_2 \land \neg t_3, t_1 \land \neg t_2 \land t_3, \neg t_1 \land t_2 \land t_3, \neg t_1 \land t_2 \land \neg t_3, \neg t_1 \land \neg t_2 \land t_3, \neg t_1 \land \neg t_2 \land \neg t_3\} \).

Since any Boolean expressions can be equivalently transformed into a disjunction of selected minterms and minterms are pair wise disjoint, a Boolean expression can be equivalently transformed to a disjunction of minterms and it is a UDNF.

Example 4: A formula \( t_1 \lor t_2 \) for the set of keywords \( \{t_1, t_2\} \) is equivalent to \( (t_1 \land \neg t_2) \lor (t_1 \land t_2) \lor (\neg t_1 \land t_2) \lor ((\neg t_1 \land \neg t_2) \lor (t_1 \land \neg t_2) \lor (\neg t_1 \land t_2)) \), which is a disjunction of minterms.

4.2. Generation of UDNF
In this section, we introduce a method to generate a UDNF for a given Boolean expression. Algorithm 1 describes a normal method to generate a UDNF of an initial query. NormGenUDNF(\( A, B \)) returns a UDNF of \( B \) where \( A \) is the universal set of keywords.
Unfortunately, however, it is not feasible to predefine a whole set of minterms in a keyword-based query model. Therefore, the result cache is represented by minterms over the set of keywords that have occurred in previous queries. Upon processing of a new query, the set of keywords is incrementally updated with new keywords from that query. Both the content and the representation of the cache are also updated. Algorithm 2 describes an incremental method to generate a UDNF of an initial query. IncGenUDNF($a$, $C$) updates the current UDNF, denoted by $C$, with a new keyword $a$.

It is well known that normalization of a Boolean expression is very expensive and often leads to an exponential increase with the size of the expression. To solve this problem in this paper, a representation of the exponential increase with the size of the expression. To solve this problem in this paper, a representation of the current result cache is semantically partitioned to several sub-representations so that the size of the representation might be significantly reduced. Section 5 covers this solution.

Algorithm 1. Normal Generation of UDNF

NormGenUDNF($A$, $B$)
Input $A$: set of keywords
$B$: target Boolean expression
Output UDNF of $B$ considering $A$
BEGIN
1. Convert $B$ to the corresponding postfix form.
2. $U$: set of minterms that can be generated by $A$
3. for every literal $l_i$ of $B$
4. convert $l_i$ to the disjunction of appropriate minterms of $U$
5. end for
6. $R$ ← minterms of the first left literal
7. $lr$: minterms of the second argument of $op$
8. if $op$ is ‘AND’ then
9. $R$ ← $R$ $\cap$ $lr$
10. else $op$ is ‘OR’ then
11. $R$ ← $R$ $\cup$ $lr$
12. end if
13. end for
14. return $R$
END

Algorithm 2. Incremental Generation of UDNF

IncGenUDNF($a$, $C$)
Input $C$: current UDNF
$a$: new keyword
Output new UDNF of $C$
BEGIN
1. for every conjunct $c_i$ of $C$
2. $c_i$ ← $c_i$ $\land$ $a$
3. $C$ ← $C$ $\cup$ $c_i$
4. $c_i$ ← $c_i$ $\land$ $\neg a$
5. $C$ ← $C$ $\cup$ $c_i$
6. end for
7. return $C$
END

4.3. Decomposition of a query to OHQ and OMQ and retrieval of results

In this subsection, we introduce a method to decompose a query into two sub-queries such as OHQ and OMQ. A minterm satisfies the following trivial properties.

Property 1: Let $\{m_1, m_2, ..., m_n\}$ be the universal set of minterms defined by a set of keywords. Suppose $Q$ is a Boolean expression and represented by $m_1 \lor m_2 \lor ... \lor m_k$ where $k \leq n$. If a document satisfies $Q$, it satisfies only one $m_i (1 \leq i \leq k)$ satisfies it but not the other $m_j$s.

Property 2: If $Q$ is a Boolean expression, then $\neg Q_{UDNF}$ is logically equivalent to the disjunction of minterms that does not occur in $Q_{UDNF}$.

According to above two properties, a mediator can easily decompose a query to its OHQ and OMQ. Suppose the current query is $Q$ and the current result cache is $C$. And suppose $Q_{UDNF}$ and $C_{UDNF}$ are UDNFs of $Q$ and $C$ respectively. In the first place, $Q \land C$ is equivalent to the set of common minterms of $Q_{UDNF}$ and $C_{UDNF}$, since other minterms that occur either in $Q_{UDNF}$ or $C_{UDNF}$ cannot satisfy $Q \land C$ according to property 1. Secondly, $Q \land \neg C$ is equivalent to the set of minterms that occurs in $Q_{UDNF}$ but not in $C_{UDNF}$, since $\neg C$ is equivalent to $\neg C_{UDNF}$ and $\neg C_{UDNF}$ is logically equivalent to the disjunction of minterms that does not occur in $C_{UDNF}$ according to property 2. Consequently, OHQ and OMQ can be computed by simply comparing minterms of $Q_{UDNF}$ and $C_{UDNF}$. (See figure 1)

![Figure 1. Compute OHQ and OMQ](image-url)

Algorithm 3 describes the procedure to generate OHQ and OMQ of the current query for the current result cache.
Example 5: Suppose a result cache is represented by (t₁ ∧ t₂) ∨ (t₁ ∧ ¬t₂) and the current query is (t₁ ∧ ¬t₂) where t₁ is a keyword. Since the query contains a keyword that does not occur in the current cache representation, the cache expression is updated to (t₁ ∧ t₂ ∧ t₃) ∨ (t₁ ∧ t₂ ∧ ¬t₃) ∨ (t₁ ∧ ¬t₂ ∧ t₃) ∨ (t₁ ∧ ¬t₂ ∧ ¬t₃) by applying IncGenUDNF(t₁, (t₁ ∧ ¬t₂)). And the query is transformed to (t₁ ∧ t₂ ∧ t₃) ∨ (t₁ ∧ t₂ ∧ ¬t₃) by applying IncGenUDNF(t₂, (t₁ ∧ t₃)). Consequently, (t₁ ∧ t₂ ∧ t₃) ∨ (t₁ ∧ t₂ ∧ ¬t₃) is an OHQ and the OMQ is null.

4.4. ‘NOT’ Operator

The following theorem makes it possible to retrieve the results of the OMQ without using negations. This is useful when the target data source does not support the ‘NOT’ operator.

Lemma 1: Suppose Q = m₁ ∨ m₂ (i ≠ j) where m₁ and m₂ are minterms defined on the set of keywords. Let n₁ (p₁) be the sub-conjunct of negative (positive, resp.) literals of mᵢ and A(n₁) be the set of atoms in n₁. If A(n₁) ⊄ A(n₂) and A(n₂) ⊄ A(n₁), then Q is logically equivalent to (p₁ ∨ p₂) ∧ (n₁ ∨ n₂).

Proof: Since m₁ = p₁ ∧ n₁ and m₂ = p₁ ∧ ¬n₁, m₁ ∨ m₂ = (p₁ ∧ n₁) ∨ (p₁ ∧ ¬n₁) and (p₁ ∨ p₂) ∧ (n₁ ∨ n₂) = (p₁ ∧ n₁) ∨ (p₁ ∧ n₂) ∨ (p₁ ∧ ¬n₁) ∨ (p₁ ∧ ¬n₂).

For p₁ ∧ n₁ (or p₁ ∧ n₂), there exist two cases.

(1) A(n₁) ∩ A(n₂) = ∅

In this case, there should exist at least one keyword that occurs in p₁ in non-negated form and in n₁ in negated form. Consequently, p₁ ∧ n₁ is always false.

(2) A(n₁) ∩ A(n₂) ≠ ∅

Since both A(n₁) - A(n₂) and A(n₂) - A(n₁) are not empty, there exists at least one keyword that occurs in p₁ in non-negated form and in n₁ in negated form. Therefore, p₁ ∧ n₁ is false in the first case.

From (1) and (2), every p₁ ∧ n₁ (or p₁ ∧ n₂) should be false. Therefore, (p₁ ∨ p₂) ∧ (n₁ ∨ n₂) is equivalent to (p₁ ∧ n₁) ∨ (p₁ ∧ n₂). Since m₁ is (p₁ ∧ n₁) and m₂ is (p₁ ∧ n₂), Q is equivalent to (p₁ ∨ p₂) ∧ (n₁ ∨ n₂).

Let N(mᵢ) be the number of literals that occur in mᵢ. Then, we can state the following lemma.

Lemma 2: Suppose Q = m₁ ∨ m₂ (i ≠ j) where m₁ and m₂ are minterms defined on the same set of keywords. If A(n₁) ⊄ A(n₂) and N(nᵢ) = N(nⱼ) = 1, then Q is logically equivalent to (p₁ ∨ p₂) ∧ (n₁ ∨ n₂).

Proof: Since A(n₁) ∇ A(n₂) and A(n₂) ∇ A(n₁), then Q is logically equivalent to (p₁ ∨ p₂) ∧ (n₁ ∨ n₂) (or (p₁ ∨ p₂) ∧ (n₁ ∨ n₂)).

The following theorem presents a necessary condition for computing a subsuming query of an OMQ.

Theorem 1: Let Q_UDNF be the UDNF of a given query Q. Formally, Q_UDNF can be defined as follows: Q_UDNF = m₁ ∨ m₂ ∨ ... ∨ mₙ where mᵢ is a minterm defined under a set of keywords. For any two minterms mᵢ and mⱼ (i ≠ j), if either A(nᵢ) ⊄ A(nⱼ) and A(nⱼ) ⊄ A(nᵢ) or A(nᵢ) ⊄ A(nⱼ) and N(nᵢ) = N(nⱼ) = 1, then Q is logically equivalent to (p₁ ∨ p₂ ∨ ... ∨ pₙ) ∧ (n₁ ∨ n₂ ∨ ... ∨ nₙ).

Proof: Q = (p₁ ∨ n₁) ∨ (p₂ ∨ n₂) ∨ ... ∨ (pₙ ∨ nₙ). And (p₁ ∨ p₂ ∨ ... ∨ pₙ) ∧ (n₁ ∨ n₂ ∨ ... ∨ nₙ) = (p₁ ∧ n₁) ∨ (p₂ ∧ n₂) ∨ ... ∨ (pₙ ∧ nₙ).

For those i ≠ j such that A(nᵢ) ⊄ A(nⱼ) and A(nⱼ) ⊄ A(nᵢ), p₁ ∧ n₁ and p₁ ∧ n₂ are false according to lemma 1. And for i ≠ j such that A(nᵢ) ⊄ A(nⱼ) and N(nᵢ) = N(nⱼ) = 1, p₁ ∧ n₁ is false and p₁ ∧ n₁ is logically equivalent to p₁ ∧ n₁ according to lemma 2. Consequently, Q is logically equivalent to (p₁ ∨ p₂ ∨ ... ∨ pₙ) ∧ (n₁ ∨ n₂ ∨ ... ∨ nₙ).
If an OMQ satisfies the conditions mentioned in theorem 1, a new query that subsumes the OMQ and does not contain any 'NOT' operator can be generated easily.

**Example 5:** Suppose an OMQ is \((a \land b \land c) \lor (\neg a \land b \land c) \lor (a \land b \land \neg c)\). Since it satisfies conditions of theorem 1, it is logically equivalent to \(((a \land c) \lor (b \land c) \lor (a \land b)) \land (\neg a \land \neg b \land \neg c)\). The first part, \(((a \land c) \lor (b \land c) \lor (a \land b))\), is sent to the data collections and the second part, \((\neg a \lor \neg b \lor \neg c)\), is used to filter out the extraneous results in the integration stage.

A query that contains a minterm that does not satisfy the conditions of theorem 1 should be processed differently. Since there exist many ways to deal with this situation, we illustrate a simple one in the following example.

**Example 6:** Suppose an OMQ is \((a \land \neg b \land c \land \neg d) \lor (\neg a \land b \land c \land d) \lor (a \land \neg b \land \neg c \land \neg d) \lor (\neg a \land b \land \neg c \land d)\). This OMQ as a whole does not satisfy the conditions of theorem 1. We partition this OMQ into two sub-queries. The first sub-query is composed of the first four minterms and the remainder is the second sub-query. Then, we can apply theorem 3 to each sub-query separately. (See figure 2)

**Definition 6: Optimal Representation**

A representation of a result cache is called optimal if it can find OHQ and OMQ for all queries by simply comparing minterms.

**Example 6:** Suppose the universal set of keywords is \(\{t_1, t_2, t_3\}\) and the current result cache is represented by \((t_1 \land t_2 \land t_3) \lor (t_1 \land \neg t_2 \land \neg t_3) \lor (\neg t_1 \land \neg t_2 \land t_3)\). Then, it is an optimal representation.

In general, if a result cache is represented by a set of minterms defined by universal set of keywords, it is always optimal. However, it is impractical since the number of keywords can be very large. To overcome this problem, we introduce a currently optimal representation.

**Definition 7: Currently Optimal Representation**

If a representation of a result cache can simply find OHQ and OMQ for the current query, it is called a currently optimal representation for the query. Every optimal representation is currently optimal for all queries.

**Example 7:** Suppose the current query is \((t_1 \lor t_2 \lor t_3)\). If the current result cache is represented by \((t_1 \land t_2 \lor \neg t_3)\) is not currently optimal. But if the same result cache if represented by \((t_1 \land t_2 \land t_3) \lor (\neg t_1 \land t_2 \land t_3) \lor (t_1 \land t_2 \land \neg t_3) \lor (\neg t_1 \land t_2 \land \neg t_3)\), it is currently optimal.

When a new query is issued, the current representation of a result cache can be transformed to a new one incrementally by applying algorithm 2 so that it is currently optimal. However, the complexity of this algorithm is \(O(2^n)\) where \(n\) is the number of current keywords, which is impractical.

### 5.2. Semantic partitioning

If we partition the set of keywords of a result cache into several sub-sets, the total number of minterms needed to describe a result cache is significantly reduced. For instance, suppose the number of keywords is 30. If we want to represent a result cache at a time, we need \(2^{30} = 1,073,741,824\) minterms. However, if we partition the set of keywords into 5 sub-sets, each set contains only 6 keywords and \(2^6 = 64\) minterms are needed respectively.

Let the current representation of a result cache \(C\) be \(R_c(K)\) where \(K\) is the set of keywords that mentioned in queries until then. \(R_c(K)\) is described by a disjunction of minterms defined by \(K\). For instance, suppose \(K\) is \(\{t_1, t_2, t_3\}\), then \(R_c(K)\) is described by \((t_1 \land t_2) \lor (\neg t_1 \land t_3)\). Some data in a result cache are identified by \((t_1 \land t_2)\) and the others by \((\neg t_1 \land t_3)\).

Suppose \(K\) is partitioned into \(K_1, K_2, ..., K_n\) where \(K\) equals to \(K_1 \cup K_2 \cup ... \cup K_n\). Then, \(R_c(K)\) can also be
partitioned into $R_1(K_1), R_2(K_2), \ldots, R_n(K_n)$. And $R_C(K)$ is equivalent to $R_1(K_1) \lor R_2(K_2) \lor \ldots \lor R_n(K_n)$. When a representation $R_C(K)$ is partitioned into $R_1(K_1) \lor R_2(K_2) \lor \ldots \lor R_n(K_n)$ and $R_C(K_i)$ ($i \neq j$) may not be necessarily disjoint with each other. (See figure 3)

![Figure 3. Semantic Partitioning of a Result Cache](image)

There exist many ways that a representation of a result cache is partitioned into some sub-representations. For example, it can be done when a metrics such as the number of conjuncts holds a specific condition. Suppose the current result cache is represented by $(t_1 \land t_2 \land \ldots \land t_j) \lor (\neg t_1 \land t_2 \land \ldots \land t_j) \lor (t_1 \land t_2 \land \ldots \land \neg t_j) \lor (t_1 \land \neg t_2 \land \ldots \land t_j)$ and the number of conjuncts are restricted not exceeding 3. Then, the current representation should be simply partitioned into two sub representations such as $(t_1 \land t_2 \land t_3) \lor (\neg t_1 \land t_2 \land \neg t_3)$ and $(\neg t_1 \land t_2 \land \neg t_3) \lor (t_1 \land \neg t_2 \land \neg t_3)$. The detailed ways to partition a result cache semantically is left to a future work.

**Definition 8: Partially Optimal Representation**

Suppose the current query is transformed to its CNF (Conjunctive Normal Form), namely, $d_1 \land d_2 \land \ldots \land d_m \land \neg d_i$, ($1 \leq i \leq m$). A partition of $R_C(K)$, $R_1(K_1) \lor R_2(K_2) \lor \ldots \lor R_n(K_n)$, is called partially optimal if there exists at least one partition $R_C(K_j)$ ($1 \leq j \leq n$) that is currently optimal for at least one of all disjuncts, $d_s$.

If a partition of $R_C(K)$ is partially optimal, a mediator can find OHQ and OMQ for the part or the current query by simply comparing minterms. Since the others can be used as a filter condition and a query contains at most 3 or 4 keywords simultaneously, a partially optimal representation is a practical solution.

### 5.2. Query processing with a partitioned cache

In this section, we introduce a method based on the partially optimal representation mentioned previous section to process general Boolean queries using a semantically partitioned cache. In the first place, methods to process single and double keyword queries are presented. Then, presented is a method to process general Boolean queries.

**Method 1: Single-keyword queries**

First of all, we consider a single keyword query. There are two possible ways to process a single keyword query with a partitioned cache. If there exists at least one sub-representation in which the keyword of a query occurs, then the query can be processed in normal way proposed in this paper. We define this way as the normal processing way. If the keyword does not occur in any sub-representation, then the query can be processed by extending any one of sub-representations. This is defined as the extended processing way.

**Method 2: Disjunctive queries with double keywords**

For a disjunctive query with two keywords, we can calculate its results by applying following properties. Let OHQ($q$, $C$) and OMQ($q$, $C$) be an optimal hit query and an optimal miss query of $q$ for a cache $C$ respectively. Then, OHQ($t_1 \lor t_2$, $C_1 \lor C_2$) and OMQ($t_1 \lor t_2$, $C_1 \lor C_2$) are logically equivalent to OHQ($t_1 \lor t_2$, $C_1 \lor C_2$) and OMQ($t_1 \lor t_2$, $C_1 \lor C_2$) respectively. Therefore, the result of ($t_1 \lor t_2$) can be answered by [OHQ($t_1 \lor t_2$, $C_1$)] [OMQ($t_1 \lor t_2$, $C_1$)] [OMQ($t_1 \lor t_2$, $C_2$)] [OMQ($t_1 \lor t_2$, $C_2$)].

In this case, there are possible three sub-cases.

**Case 1: A keyword $t_i$ occurs in a $K_i$**

In this case, we can process the query directly by means of above property.

**Case 2: One keyword $t_i$ occurs in a $K_i$ but not the other**

Suppose $t_i$ occurs in $K_i$ but $t_j$ does not occur in any sub-representation. Then, OHQ($t_i$, $C_i$) and OMQ($t_i$, $C_i$) can be calculated in normal processing way. OHQ($t_j$, $C_i$) and OMQ($t_j$, $C_i$) can be calculated in extended processing way.

**Case 3: Both of two keywords do not occur in any sub-cache**

Every OHQ and OMQ can be calculated in an extended processing way.

**Method 3: Conjunctive queries with double keywords**

For a conjunctive query, we only have to calculate the results of one (we call this a hit keyword) of two keywords and filter out its results with the other keyword.
If there exists at least one keyword that occurs in a sub-representation, we select that keyword as a hit keyword. The result of a hit keyword can be calculated in normal processing way. If both of two keywords do not occur in any sub-representation, we select a random keyword as a hit keyword. In this case, the result of a hit query can be calculated in extended processing way.

**General queries**

A query is transformed to its CNF (Conjunctive Normal Form). Let \( Q \) be a query and \( Q_{\text{CNF}} \) be its CNF. According to method 3, we select any one disjunct from \( Q_{\text{CNF}} \) and process it by applying method 2. Then, we filter out its results with other disjuncts. The efficiency of this method is dominated by which disjunct might be selected. Since the main goal of cache-based query processing is to reduce the size of results that should be transferred from collections, we select a disjunct so that the number of keywords not in a cache should be minimal.

### 6. Performance Analysis

To prove the efficiency of our method, we use the following notation:

<table>
<thead>
<tr>
<th>Table 1. Performance Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metrics</td>
</tr>
<tr>
<td>( tM )</td>
</tr>
<tr>
<td>( tD )</td>
</tr>
<tr>
<td>( tMS )</td>
</tr>
<tr>
<td>( tS )</td>
</tr>
<tr>
<td>( nQ )</td>
</tr>
<tr>
<td>( mQ )</td>
</tr>
<tr>
<td>( nC )</td>
</tr>
<tr>
<td>( mC )</td>
</tr>
<tr>
<td>( dC )</td>
</tr>
<tr>
<td>( N )</td>
</tr>
<tr>
<td>( T(tQ) )</td>
</tr>
</tbody>
</table>

First of all, we assume that \( dC \) and \( N \) are greater than or equal to 3. This assumption is not only reasonable but also practical since it is common that the number of results for a query is large.

Since \( k \cdot tM + (N - k) \cdot tS \) is the time to retrieve \( k \) results from a cache and \( (N - k) \) from a search engine, the expected time to retrieve whole results from both a cache and a search engine should be \( \frac{1}{N} \sum_{k=0}^{N-1} (k \cdot tM + (N-k) \cdot tS) \). Therefore, \( tMS \) can be calculated as follows:

\[
tMS = \frac{1}{N^2} \sum_{k=0}^{N-1} (k \cdot tM + (N-k) \cdot tS)
\]

\[
= \frac{1}{N^2} \left( \sum_{k=0}^{N-1} k \cdot tM + \sum_{k=0}^{N-1} (N-k) \cdot tS \right)
\]

\[
= \frac{1}{N^2} \left( (tM \sum_{k=0}^{N-1} k + \sum_{k=0}^{N-1} N \cdot tS - N \sum_{k=0}^{N-1} k) \right)
\]

\[
= \frac{1}{N^2} \left( \frac{N(N-1)tM}{2} + N(N-1)tS - \frac{N(N-1)tS}{2} \right)
\]

\[
= \frac{1}{N^2} \left( \frac{N(N-1)tM + N(N-1)tS}{2} \right)
\]

\[
= \frac{N(N-1)(tM + tS)}{2N^2}
\]

**Theorem 2:** For all queries, \( tM > tD \).\(^1\)

**Proof**

From algorithm 3, we can say the following:

\[ tD = 2^{nC+mQ} + 2^{2nQ+mc} + 2^{nC + mQ + nQ + mC} \]

In order to retrieve data from the cache, we should divide each partition of the cache into sub-partitions by means of newly-occurred atoms in a query. Then, compare the conjuncts of the cache with those of the query. Since the average number of data objects in a conjunct of the cache is \( dC \) and the number of atoms in the query and the cache is \( nC + mQ \), it takes \( dC \cdot 2^{nC+mQ} \) to retrieve data from the cache. It also takes \( 2^{nC+mQ+nQ+mC} \) to compare the conjuncts of the cache with those of the query. Consequently, we can say the following:

\[ tM = dC \cdot 2^{nC+mQ} + 2^{nC+mQ+nQ+mC} \]

Therefore,

\[ tM - tD = (dC-1)2^{nC+mQ} - 2^{nC+mQ+(1+\frac{1}{2^{nC}})} \]

Since \( nC + mQ = nQ + mC \)

\(^1\) In fact, \( tM > tD \) can be trivially proved since we assumed that \( tD \) is contained in \( tM \). However, we present a proof to show this assumption is feasible.
$tM - tD = (dC - 2 - \frac{1}{2m})^2 n^{C+mQ}$

$> 0(\because dC > 3)$

$\therefore tM > tD \blacksquare$

According to theorem 2, it is clear that retrieving results from the cache is more expensive than decomposing a query into OHQ and OMQ. In other words, it is not too expensive to decompose a query.

In general, $tM$ is much larger than $tS$ since $tS$ contains a network overhead and a query should be sent to many diverse data sources. If this assumption is agreed on, we can state the following theorem:

**Theorem 3:** For all queries, $tS > tMS > tM$.

*Proof*

I) $tS > tMS$

$tS - tMS = \frac{N(N-1)(tM + tS)}{2N^2}$

$= \frac{(N^2 - N)tS - (N^2 - N)tM - (N^2 - N)tS}{2N^2}$

$= \frac{(N^2 + N)tS - (N^2 - N)tM}{2N^2}$

$= \frac{(N^2 + N)tS - (2N)tM}{2N^2}$

$> 0(\because N > 0, tS > tM)$

II) $tMS > tM$

$tMS - tM = \frac{N(N-1)(tM + tS)}{2N^2} - tM$

$= \frac{(N^2 - N)tM + (N^2 - N)tS - 2N^2 \cdot tM}{2N^2}$

$= \frac{(N^2 - N)tS - (N^2 + N)tM}{2N^2}$

$= \frac{(N^2 - N)tS - (2N)tM}{2N^2}$

$= \frac{(N^2 - N)(a \cdot tM - tM) - 2N \cdot tM}{2N^2}$

$= \frac{((N-1)(a-1) - 2)N \cdot tM}{2N^2}$

$((N-1)(a-1)-2)$ should be greater than zero to make $tMS > tM$. Since $N \geq 3$ and $N - 1 \geq 2$, $a$ should be greater than 2 to make $(N-1)(a-1) > 2$. Since it is common that the time to retrieve data from a search engine is much greater than that to retrieve data from a cache, we can reasonably and practically assume that $a$ is always greater than 2. Therefore, from I and II, $tS > tMS > tM$. $\blacksquare$

[10] has reported that on the average 88% of the total queries uses terms that have been used already. So, we can state that a keyword repeatedly occurs in a new query with probability $p$.

**Theorem 4:** For a given query $Q$, $T(Q)$ is always less than $tS$.

*Proof*

$Q$ can be transformed a corresponding DNF like $C_1 \lor C_2 \lor ... \lor C_n$ where $C_i$ is $(t_{i1} \land t_{i2} \land ... \land t_{im})$, $t_i$ is a keyword. There exist three cases in processing $C_1 \lor C_2 \lor ... \lor C_n$.

1) The result of $C_1 \lor C_2 \lor ... \lor C_n$ can be retrieved only from the cache. In this case, we don’t have to send $C_1 \lor C_2 \lor ... \lor C_n$ to any search engines. In this case, every $C_i$ should be answered from the cache. We can show that the probability that $C_i$ might be answered from the cache is $(1 - (1 - p)^n)$. Therefore, the whole probability of this case should be $(1 - (1 - p)^n)^n$.

2) The result of $C_1 \lor C_2 \lor ... \lor C_n$ should be retrieved only from search engines, since the cache does not contain any results for $C_1 \lor C_2 \lor ... \lor C_n$. In this case, we should get answers of every $C_i$ from search engines only. We can show that the probability that $C_i$ might be answered only form search engines is $(1 - p)^n$. Therefore, $(1 - p)^n$ is the probability of this case.

3) If the cache can answer $C_1 \lor C_2 \lor ... \lor C_n$ partially, we can get answers of $C_1 \lor C_2 \lor ... \lor C_n$ from both the cache and search engines. The probability of this case is $1 - (1 - p)^n - (1 - (1 - p)^n)^n$, since the sum of these three probabilities should be 1.

Consequently, $T(Q)$ is $(1 - (1 - p)^n)^n \cdot tS + (1 - (1 - p)^n) \cdot tMS$. Let $(1 - p)^n$ be $q$ then, $0 < q < 1$ and $T(Q)$ is $(1 - q)^n \cdot tM + q^n \cdot tS + (1 - q^n) \cdot tMS$.

$tS - T(Q)$

$= tS - (1 - q)^n \cdot tM - q^n \cdot tS - (1 - q^n) \cdot tMS$

Since $tS > tMS > tM$, $tS$ and $tMS$ can be replaced by $a \cdot tM$ and $b \cdot tM$ respectively where $a > b > 1$. Therefore, $tS - T(Q)$

$= a \cdot tM - (1 - q)^n \cdot tM - a \cdot q^n \cdot tM - b \cdot tM$

$+ b \cdot q^n \cdot tM + b(1 - q^n) \cdot tM$

$= ((a - b)(a - b)q^n + (b - 1)(1 - q^n)) \cdot tM$

$= ((a - b)(1 - q^n) + (b - 1)(1 - q^n)) \cdot tM$

$> 0(\because a > b > 1, q < 1)$

$\therefore tS > T(Q) \blacksquare$

According to theorem 4, our method can process any Boolean queries more efficiently than others that do not use the cached results.
7. Conclusion

In this paper, we studied the management of a result cache under the keyword-based Boolean query model in the mediator context. In order to improve the query processing performance of a mediator system, it is essential to efficiently recognize which part of a given query can be answered from a result cache and which should be sent to target collections. In this paper, we proposed an efficient method that allows this by adopting the general Boolean query model in representing the query and the result cache.

Currently, we are extending the work to query models based on first order logic to allow attribute-based Boolean queries.

8. References