Using User Access Patterns for Semantic Query Caching

Qingsong Yao and Aijun An

Department of Computer Science, York University, Toronto M3J 1P3 Canada
{qingsong,aan}@cs.yorku.ca

Abstract. In this paper, we propose a solution that partly solves the selection and replacement problems for semantic query caching. We believe that the queries submitted by a client are not random. They have certain meaning and may follow certain rules. We use *user access graphs* to represent the query execution orders and propose algorithms that use such information for semantic query caching. Unlike the previous approaches, ours anticipates incoming queries based on the queries that have been submitted, analyzes the semantic relationship between them, and rewrites and caches the current query to answer multiple queries. Our initial experimental result shows that our solution improves cache performance.

1 Introduction

Semantic query caching (SQC) [3, 5, 6] assumes that the queries submitted by a client are related to each other semantically. Therefore, grouping tuples according to the semantic meaning can provide better performance than other caching approaches (i.e., page-based or tuple-based caching). Previous research considers every submitted query as a cache candidate. Such solution is not extendable since the number of cache candidates for a given client application is quite large. The cache replacement algorithms are based on the statistics (i.e., query result size, query referencing frequency or underlying data updating frequency). The replacement algorithms may not provide good performance since they do not take query locality into account. Meanwhile, in order to find and retrieve the answer of a query from caches, an algorithm is needed to examine how they are related semantically. A special case is that the query is semantically contained in the caches, which is called *query containment*. There are many studies in this field, but the execution time of the algorithms cannot be ignored when the number of caches is large.

In this paper, we use *user access patterns* to describe how a user or a client accesses the data of a database, and explore ways to use such information for semantic query caching. We can anticipate the queries to be submitted by using the query execution patterns. We found that a user always accesses the same part of the data within certain time interval. Thus, semantic relationships exist between the corresponding queries, and can be used to rewrite and cache a query

to answer multiple queries. The rewritten queries save both the network load and the processing cost of the server.

The rest of the paper is organized as follows. User access patterns and related issues are presented in Sect. 2. In Sect. 3, we propose three caching solutions: *sequential-execution*, *union-execution* and *probe-remainder-execution*. We present the evaluation strategies and experimental results in Sect. 4. We describe related work in Sect. 5. Sect. 6 is the conclusion.

2 User Access Patterns

We can transform a SQL query into an *SQL template* and a set of *parameters*. We treat each integer value or string value of a given SQL query as a parameter, and the SQL template can be obtained by replacing each parameter in the SQL with a wildcard character (%). Database users often submit *similar queries* to retrieve certain information from the database. We use a *user access event* to represent a set of similar queries which share the same SQL template. A user access event contains an SQL template and a set of parameters.

The query execution order, represented by a directed graph, is called a *user access graph*. The graph has one start node and one or many end nodes, and cycles may exist in a graph. Each node is a user access event. An edge is represented by $e_k : (v_i, v_j, \sigma_{v_i \to v_j})$, where $\sigma_{v_i \to v_j}$ is the probability of v_j following v_i, which is called the confidence. The graph associates with a support value τ_g which describes how often it is executed[1]. For example, table 1 lists an instance of a patient information model which retrieves a given customer's information. Fig. 1 shows the corresponding user access graph. The graph is a special graph since it contains no branches or cycles. It is called a *user access path*.

Table 1. An instance of patient information model

Label	Statement
30	select authority from *employee* where employee_id =**'1025'**
9	select count(*) as num from *customer* where cust_num = **'1074'**
10	select card_name from customer t1,member_card t2 where t1.cust_num = **'1074'** and t1.card_id = t2.card_id
20	select contact_last,contact_first from *customer* where cust_num = **'1074'**
47	select t1.branch ,t2.* from *record* t1, *treatment* t2 where t1.contract_no = t2.contract_no and t1.cust_id =**'1074'** and check_in_date = **'2002/03/04'** and t1.branch = **'scar'**

The graph may contain a set of global variables: employee id (*g_eid*), customer id (*g_cid*), branch id(g_*bid*) and check-in date (*g_date*). Variable *g_date*

[1] The support and the confidence have the same meaning in the field of association rule mining.

is a system variable which equals to today's date, and we call it as a constant global variable. Meanwhile, *g_cid* is unknown before event *v9* is submitted, so the parameter of *v9* is a local variable(*l_cid*). When *v9* is submitted, *g_cid* is set to the value of *l_cid*, then event *v10,v20* are determined since their parameters are known. Thus, some nodes in the graph associate with a set of actions which change the value of global variables according to the values of its local variables.

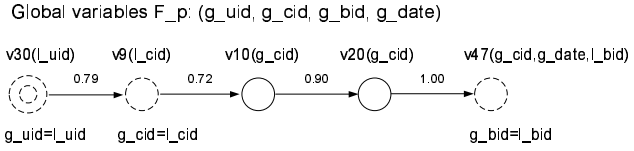

Fig. 1. User access path for patient information module

There are two kinds of events: *determined event* and *undetermined event*. An event is a determined event if and only if each parameter is either a constant or a global variable. Otherwise, it is undetermined. Three kinds of determined events, graph-determined event, parameter-determined event and result-determined event, exist in a graph. If every parameter of an event is a constant or a constant global variable, we call it a graph-determined event. Event v_j is result-determined by v_i, if and only if one or more parameters of v_j are unknown until v_i finishes execution[2]. Event v_j is parameter-determined by v_i, if and only if one or more parameters are unknown until v_i is submitted. For example, *v30*, *v9* and *v47*, represented by dotted cycles in the graph, are undetermined events, and *v10* and*v20* are parameter-determined by *v9*.

We use *user access patterns* to describe how a client application or a group of users access the data of a database. User access patterns include a collection of *frequent* user access graphs whose support is bigger than a given threshold τ, and a collection of user access events associated with the occurrence frequency and the parameter distribution. In this paper, we assume that the user access graphs are already obtained from database workload files or from the corresponding business logic, and the graphs are broken into several user access paths which can be processes efficiently. We are interested in finding and using the semantic relationship between the events of the user access path.

3 SQC Selection and Replacement

3.1 SQC Selection Problem

The SQC selection problem is defined as follows. Given a user access path p and a cache pool of size δ, find a set of query rewriting rules (for semantic query

[2] The difference between an undetermined event and a result-determined event is that the parameters of the latter can be derived from the query result of other queries.

caching) to minimize the total execution cost of p. During the execution of the queries of path p, the size of the cached data should not exceed the cache size δ. In this section, we call a user access query a parameterized query, referred to as a query.

We first consider to generate rewriting rules for two consecutive queries u and v, and then extend the solutions to the whole path. We assume that the queries are *SPJ (select-project-join)* queries which can be written in the form $\{\alpha, \gamma, \delta\}$, where $\alpha = \{\alpha_1, \alpha_2, ...\alpha_m\}$ is a set of *selection attributes*, $\gamma = \{\gamma_1, ..\gamma_k\}$ is a set of relations and δ is the predicate. The attributes involved in the query predicate are called *search attributes*.

We use R_u to represent the answer of u. When query u is submitted, three solutions can be used to retrieve R_u and R_v. The first solution is to pre-fetch R_v, which is called the *sequential-execution* (*SEQ*) solution since we execute u and v sequentially. We may also submit the union query $(u \cup v)$ to answer both u and v together, and it is called the *union-execution (UNI)* solution. In order to retrieve the answer of the two queries from the cached data, the union query must include both the selection attributes of them and some search attributes. The third solution is called *probe-remainder-execution (PR)* solution. In this solution, an extended version of query u, referred as to u', is submitted and cached which includes columns needed by query v. To answer v, the solution retrieves part of the answer from $R_{u'}$, as well as submitting a remainder query v' to the server to retrieve the tuples which are not in the cache. The *UNI* and *PR* solutions are not applicable when two queries do not access the same relations.

The *SEQ* solution pre-executes queries to shorten the latency between the request and the response, while the *UNI* and *PR* solution aim to improve response time by decreasing the network transmission cost and the server processing cost. The *SEQ* has extra network transmission and server processing cost when query v isn't submitted. The *UNI* may improve server performance by accessing the base relation only once, and the *PR* may cause the server to access less data. But, the *UNI* and *PR* solutions may introduce *"disjunction"* and *"negation"* operations respectively which may cause the server to generate different query execution plans and increase the server processing cost. The *SEQ* and *UNI* solutions have extra network transmission cost when query v isn't submitted, but the *PR* solution may save the cost by retrieving part of the query answer from the caches.

3.2 Semantic Relationship Between Parameterized Queries

We use the similar idea presented in [5] to illustrate the possible semantic relationships between two parameterized queries u and v in Fig. 2. Each query result can be viewed as an abstract relation which is represented by a box in the figure. The x-axis corresponds to the columns or the selection attributes, and the y-axis corresponds to the corresponding rows of the original relations. We are interested in the relationship between the rows since we can rewrite queries to include more columns. We summarize these cases as:

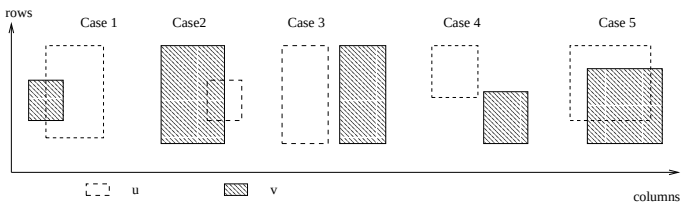

Fig. 2. Semantic relationship between two parameterized queries

- *containing-match.* In case 1, the rows of u contains those of v. The union and the probe query are both u', and there is no remainder query. Thus, the *UNI* solution and the *PR* solution become the same. The solution usually does not generate worse query execution plans compared to the original queries, and the overall result size determines whether the solution is applicable.
- *contained-match.* In case 2, the rows of v contain those of u. The *UNI* and *PR* solutions are the possible solutions. When $\sigma_{u \to v}$ is small, we choose *PR* solution since *UNI* solution has extra costs when v isn't submitted.
- *horizontal-match (h-match).* In this case, u and v have the same set of rows, but may have different columns. The *UNI* and the *PR* solution usually do not generate worse query plans and have better query performance than the *SEQ* solution. When v has more columns than u and $\sigma_{u \to v}$ is small, we choose the *PR*. Otherwise, we choose *UNI*.
- *u disjoint with v.* In this case, there is no common rows between R_u and R_v (case 4). The *PR* solution is not suitable, but the *UNI* solution is applicable since it may save the server cost by only accessing the base relation once.
- *u partial-match with v.* That is, R_u and R_v have common rows (case 5). All three solutions may be applicable in this case.
- *u and v are irrelevant.* Two queries are irrelevant if their base relations are different, and the only possible solution is *SEQ*.

3.3 Algorithms

In this section, we propose an off-line algorithm, *rewriting-one*, to solve the SQC selection and replacement problem. The algorithm tries to find the semantic relationship between u and v, builds up rewriting queries and chooses an optimal rewriting query based on the costs. The *rewriting_one* algorithm is described in Figure 3.

We assume that the predicates of u and v are both the conjunction of basic predicate units which have the following format: *"var op constant"* or *"var op var + constant"*, where the operators are $\{=, <, >, \leq, \geq\}$ and the domain of each variable is integer. The algorithm first transforms a query predicate into an expression that only contains $\leq$ operators, and builds a weighted directed graph which describes the range of the search attributes and the relationship between

Algorithm: rewriting_one(u,v).
Input: two queries u and v.
Output: query rewriting solution.

1. $solutions = \{\}$;
2. if v is *undetermined*, $solutions = \{\}$;
3. else if v is *result-determined* by u, $solutions = \{SEQ\}$;
4. else if u and v are *irrelevant*, solutions = {SEQ};
5. else $G_u = weighted\_directed\_graph(\delta_u)$;
6. $G_v = weighted\_directed\_graph(\delta_v)$;
7. $comm\_diff(G_u, G_v, comm, diff)$;
8. $rel = relation(comm, diff)$;
9. switch (rel):
10. case *containing*:$solutions = \{UNI\}$;
11. case *contained,h_match*:$solutions = \{UNI, PR\}$;
12. case *disjoint*: $solutions = \{SEQ, UNI\}$;
13. case *partial-match*: $solutions = \{SEQ, UNI, PR\}$
14. $build\_queries(\alpha_u, \alpha_v, comm, diff)$;
15. select a solution based on the overall costs.

Fig. 3. Algorithm rewriting_one

them[3]. An edge $e(x, y, c)$ of the graph means that $x \leq y + c$ holds. For example, given queries:

- u1: *"select a1 from r1 where a2=1 and a3≤3"*
- u2: *"select a1 from r1 where a2=1"*
- u3: *"select a1 from r1 where a2=1 and a3≥1"*,

and we want to generate query rewriting rules for query pair *u1* and *u2*, and for query pair *u2* and *u3*. The graphs are listed in Figure 4, where the dotted edges are marked as *derived* since they are derived from other edges.

In step 7, the algorithm builds the common edges set (*comm*) and the different edges set (*diff*) between two graphs. For each edge in G_u, the algorithm tries to find the corresponding edge in G_v, or vice versa. If the weights of two edges are the same, we add them to *comm* set, otherwise we add them to *diff* set. For example, *u1* and *u2* have the same predicate unit *a2=1* which corresponds to two identical edges in the graphs. In step 8 of the algorithm, we find the relationship between two queries by testing each pair of edges in the *diff* set. If two queries have *h-match* relationship, the *diff* set is empty. If they have *contained-match* relationship, then for each edge pair $< e_u, e_v >$ in the *diff* set, either e_v is *null* or e_v has a larger weight than e_u. The *containing-match* can be tested in a similar way. The *disjoint-match* is obtained by testing whether $u \cap v$ has an answer, which in turn tests whether there is a negative circle in the graph of $u \cap v$. In step 9, we convert the edges of the *diff* and the *comm* set

[3] Weighted directed graph was first introduced in [8]

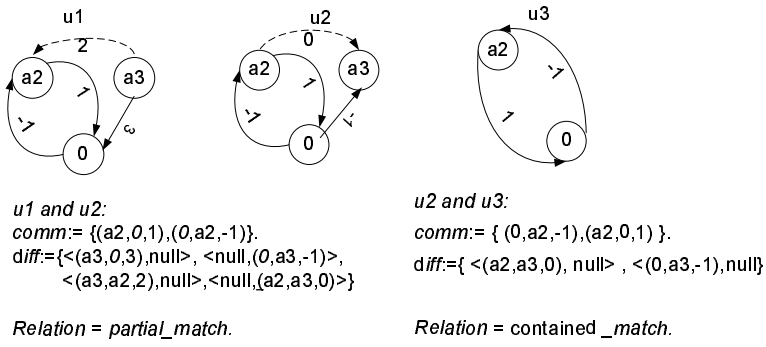

Fig. 4. Weighted directed graphs for query u1, u2, and u3

into corresponding predicate units, and use logic conjunction, disjunction and negation operators to connect them.

In the previous example, *u1 partial-matches* with *u2*, and *u2* is contained by *u3*. The rewriting queries for *u1* and *u2* are:

- union query: *"select a1,a3 from r1 where a2=1"*
- probe query: *"select a1,a3 from r1 where a2=1 and a3* $\leq$*3"*
- remainder query: *"select a1 from r1 where a2=1 and t3* $>$*3"*.

The *rewriting-one* algorithm can also process queries which contain join operations. For example, query *10* in Figure 1 is a join query, which can be rewritten as *"select card_name from member_card where card_id in (select card_id from customer t1 where t1.cust_num ='1074')"*, and the sub-query has *h-math* with queries *9* and *20*. Algorithm *rewriting-one* only considers two consecutive queries in a path, and in many cases, two consecutive queries do not have a *strong* relationship, such as *h-match*, *containing-match*, or *contained-match*. Thus, we suggest a *rewriting-n* algorithm which can generate global optimal plans by considering all queries in the path together. The algorithm finds a candidate set for every query, and analyzes the relationship between them. Then the algorithm sorts the set according to the rank of the relationships, and an optimal plan is generated by applying a heuristic method based on accumulative costs and available spaces.

4 Performance Evaluation and Experiments

We propose and implement a database proxy program *SQL-Relay*, to be the platform of our SQC solution. *SQL-Relay* is an event-driven, rule-based database proxy program which intercepts every message sent from the client or the server, and treats it as an event. It traces the user request sequence for each connected client. When the sequence matches one user access path, a set of pre-defined rewriting rules are applied. The *SQL-Relay* cache manager manages two kinds

of caches: the global caches and the local caches. All connected clients share a global cache pool which is managed by using the LRU replacement policy. Meanwhile, each client also has a local cache pool. Each local cache entry has a *reference count* which indicates how many queries can be (partly) answered currently. When the *reference count* of an entry becomes zero, it is moved to the global cache pool, and we move all local caches into the global cache pool when a path finishes execution. To maintain the caches efficiently, we use the similar idea in [6] to describe the contents of each cache entry by using a predicate description, and discard the cache when the content of it is changed.

Now, we present the experimental result of an OLTP application. A clinic office has a client application, *Danger Front*, which helps the employee check in patients, make appointments and sell products. After preprocessing one day's database queries log, we found 12 instances of the application. The query log has 9,344 SQL statements in 190 SQL templates, where 72% (136) of queries are *SPJ* queries. 718 sequences are found from the log. They belong to 21 user access paths which have support bigger than 10. We generate 19 *SEQ* rules, 4 *UNI* rules and 3 *PR* rules by using the rewriting-n algorithm.

We use *MySQL* as our database server which features a server-side *query cache* function, and compare the performance of server caching with our approach. The connection speed between the database and the *SQL-Relay* is 56 Kbps, and the speed between the clients and the *SQL-Relay* is 10 Mbps. We synthesize 300 client request sequences based on the path supports, and compare the cache performance under the following conditions: (1) executing queries without cache, (2) executing queries with 128K server cache, (3) pre-fetching queries based on user access patterns, and (4) using rules generated by using *rewriting-n*. In case 3 and 4, the caching function of *MySQL* is disabled and the *SQL-Relay* configures with a 128K cache. Each connected client has 4K local caches, and the size of global caches dramatically changes as more clients get connected or disconnected.

Table 2. Comparison of cache performance (per 100 queries).

	response time (s)	global cache hit	local cache hit	queries sent to server	network traffic(K bytes)	server I/O (K blocks)
case 1	11.4	N/A	N/A	100	61.0	1,390
case 2	11.2	19.1	N/A	100	62.1	1,182
case 3	10.8	20.2	10.5	82.6	63.7	1,207
case 4	10.3	21.3	6.4	78.2	50.6	1,108

The result is listed in Table 2. The query response time is calculated at the client side. It includes server processing time, network transmission time, cache processing and maintaining time. We found 87% of queries are selection queries which did not change the state of the server. Thus the cache maintenance cost

is not too high. The response time of our solution is the shortest since it saves both network transmission time and server processing time. The pre-fetching case has better response time than the server caching, but it doesn't improve server performance and has heavier network traffic than the latter.

Figure 5 shows the cache performance when the support of a path or the confidence of an event changes. We decrease the support of paths by increasing the frequency of random paths. The result is shown on the left of the figure. It shows that the network load decreases, and so does the local cache hits, since the paths are not frequent. Then we do not change the support of the paths, but decrease the confidence of queries in a path by increasing the frequency of the random queries. The result is shown on the right of the figure. It shows that the network load as well as the local cache hits also decrease since we cannot find suitable rules.

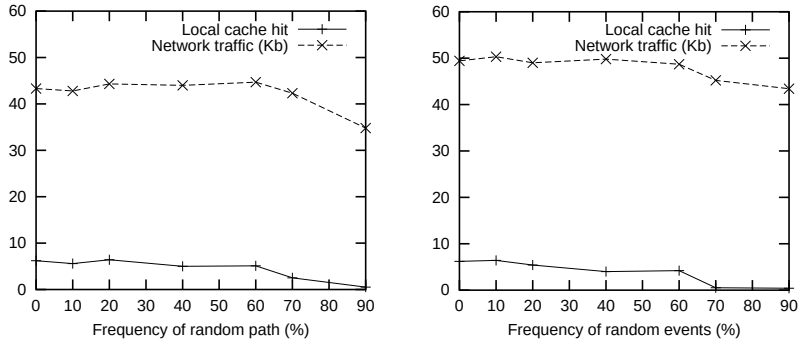

Fig. 5. Cache performance with different support or confidence. (per 100 queries)

5 Related Work

The cache selection and replacement problem has been extensively studied in [11]. Sellis proposes the cache selection algorithms for both the *unbounded space* where the available cache space is infinite and the *bounded space* where there is limited space. Sellis suggests various *rank* functions for an LRU-like replacement policy. Dar *et al* [3] use a semantic region as the unit of cache replacement. Their replacement policies can be based on temporal locality or spatial locality. Keller and Basu [6] use predicate descriptions to describe the cached data for each previously-asked query. They focus on using server-side and client-side predicate description to maintain the cached data efficiently. Godfrey and Gryz [5] provide a comprehensive logic-based framework of semantic query caching by using *Datalog*, and introduce the concepts of semantic overlaps, semantic independence and semantic query remainder.

Similar studies focus on using database access patterns to predict the buffer hit ratio [1, 2], to improve caching performance of OLAP systems [10], and to

generate global optimal query execution plans [4, 9]. In [10], Sapia discusses the PROMISE approach which provides the cache manager with access patterns to make prediction in the OLAP environment. Finkelstein [4] describes an algorithm to detect common expressions between a set of queries. Multiple query optimization [9] generates global optimal query execution plans for multiple queries. Their solutions are server-side solutions which help the server to process multiple queries, but the intermediate results may be too large to be cached by a client or a mediator.

6 Conclusion

In this paper, we propose a method that preforms semantic query caching by using user access patterns. To our knowledge, it is the first attempt to make use of such information for semantic query caching. Compared with other semantic query techniques, our SQC approach has several advantages. Our caching algorithms are based on the query execution orders and the semantic relationship between queries, which are better than the selection policies based on the global query reference statistics. The pre-defined query writing rules simplify the cache finding and replacing procedure. Our SQL-Relay application is flexible and extendable where various caching and rewriting rules can be added and tested. We would like to thank Professor Jarek Gryz for providing valuable feedback on this paper.

References

1. A. Dan, P. S. Yu, and J. Chung. Database Access Characterization for Buffer Hit Prediction. ICDE 1993: 134-143.
2. A. Dan, P. S. Yu, and J. Chung. Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability. VLDB Journal 4(1) 1995.
3. S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivatava and M. Tan. Semantic Data Caching and Replacement. Proc. VLDB Conf, 1996.
4. S. Finkelstein. Common Expression Analysis in Database Application. In Procs. of ACM SIGMOD, 1987.
5. P. Godfrey and J. Gryz. Answering Queries by Semantic Caches. DEXA 1999: 485-498.
6. A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In Proc VLDB Conf.,1996.
7. MySQL Reference Manual (*http://www.mysql.com*).
8. D. J. Rosenkrantz and H. B. Hunt. Processing Conjunctive Predicates and Queries. In Proceedings of VLDB, 1980
9. T. Sellis and S. Ghosh. On the multiple-query optimization problem. TKDE,2(2), June 1990.
10. C. Sapia. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. In Proc. DAWAK 2000, 224-233.
11. T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. Inform. Systems,13(2), 1988.