Characterizing Database User's Access Patterns

Qingsong Yao and Aijun An [*]

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

Abstract. Much work has been done on characterizing the workload of a database system. Previous studies focused on providing different types of statistical summaries, and modeling the run-time behavior on the physical resource level as well. In this paper, we focus on characterizing the database system's workload from the view of database user. We use *user access patterns* to describe how a client application or a group of users access the data of a database system. The user access patterns include a set of *user access events* that represent the format of the queries and a set of *user access graphs* that represent the query execution orders. User access patterns can help database administrators tune the system, help database users optimize queries, and help to predict and cache future queries. In this paper, we will present several approaches to use user access patterns to improve system performance, and report some experimental results.

1 Introduction

Database workload characterizing is usually based on the database traces. The traces are a collection of measures, such as pages read/written, number of locks and number of SQL statements, produced by all transactions being processed by the DBMS within a time interval. Database workload characterization can be based on the three abstract levels: application level, transaction or session level, and physical resource level [9]. Previous studies focus on providing all kinds of statistical summaries and run-time behavior on the physical resource level. Most of the work is from the view of database server. We are interested in database user behaviors in the session level, i.e., the queries submitted by a user within a user session. In this paper, we propose the idea of *user access patterns* to characterize the database's workload from the view of database users. The main components of the user access patterns are a collection of user access events which represent the format of the queries, and a collection of user access graphs which represent the query execution orders. We suggest several ways to improve system performance by applying user access patterns. The rest of the paper is organized as follows. We discuss the concepts of user access patterns and give formal definition in Section 2. In Section 3 , we propose several approaches to improve system performance by using user access patterns. We give experimental results in Section 4. Related work is discussed in Section 5, and Section 6 is the conclusion.

[*] This work is supported by research grants from Communications and Information Technology Ontario (CITO) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

2 User Access Event and User Access Graph

In this section, we propose two concepts, user access event and user access graph to represent the query format and query execution orders respectively.

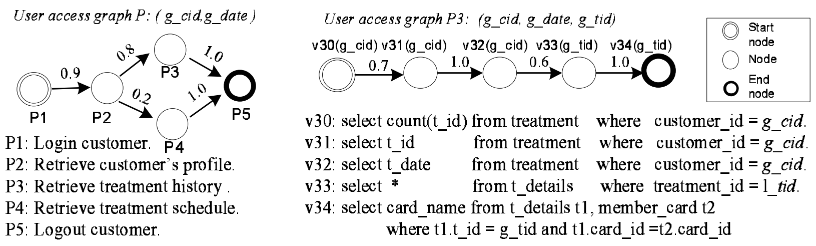
Given an SQL query, we can transform it into two parts: an *SQL template* and a set of *parameters*. We treat each data value embedded in the query as a parameter, and the SQL template is obtained by replacing each data value with a wildcard character (%). In many situations, the SQL queries submitted by a user have the same SQL template and only have different in the data value part. In this paper, we use a *user access event* to represent the queries with similar format. A user access event contains an SQL template and a set of parameters. The value of a parameter is either a constant or a variable. For example, event *("select name from customer where id =% ", 'c101')* represents a single query which retrieves the name of a customer *'c101'*, while event *("select name from customer where id ='%' ", g_cid)* represents a set of similar queries.

Definition: SQL template - the SQL template of a given query is obtained by replacing each embedded data value with a wildcard character %.

Definition: user access event - a user access event $v : (S_t, F_v)$ contains an SQL template S_t and a set of parameters F_v. $F_v = (f_1, ...f_m)$ is a set of parameters corresponding to the wildcard characters of S_t.

From the view of database user, in order to execute certain task, a sequence of queries is submitted to the server. We call such a sequence of queries as a user session. Given one set of user sessions of the same type, we use a *user access graph* to represent the corresponding query execution order. A user access graph is a directed graph that has one start node and one or more end nodes. Each node v_i has a support value ρ_{v_i}, which is the occurrence frequency within the set of user sessions, and 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 of the edge. We usually require each node and each path must have a minimal support or a confidence, in which we can eliminate noise requests in the session set.

There are two types of user access graphs depending on the granularity of the nodes. The first type is called the basic user access graph, whose nodes represent only events. The second type is a high level user access graph, in which each node is represented by a basic user access graph. A high level user access graph is used to describe the execution orders among sessions that are performed by the same user.


User access graph P: (g_cid,g_date)
0.9
0.8
0.2
1.0
1.0
P1
P2
P3
P4
P5
User access graph P3: (g_cid, g_date, g_tid)
v30(g_cid) v31(g_cid) v32(g_cid) v33(g_tid) v34(g_tid)
0.7
1.0
0.6
1.0
Start node
Node
End node

P1: Login customer.
P2: Retrieve customer's profile.
P3: Retrieve treatment history .
P4: Retrieve treatment schedule.
P5: Logout customer.

```
v30: select count(t_id) from treatment   where  customer_id = g_cid.
v31: select t_id         from treatment   where  customer_id = g_cid.
v32: select t_date       from treatment   where  customer_id = g_cid.
v33: select  *           from t_details    where  treatment_id = l_tid.
v34: select card_name from t_details t1, member_card t2
              where t1.t_id = g_tid and t1.card_id =t2.card_id
```

Fig. 1. An example of user access graphs

Figure 1 shows two-level user access graphs for the patient-information model. When a user logs in (P1), the corresponding profile is retrieved (P2). Then he/she can retrieve either treatment history (P3) or treatment schedule (P4). Graph *P*, illustrated on the left side, is a high-level user access graph that describes the execution orders of five sub-models. Table 1 is an instance of the treatment-history sub-model which contains five consecutive SQL queries. The corresponding user access graph *P3* is shown on the right side of Figure 1. We call a the user access graph with nodes that are user access events a basic user access graph.

The patient-information model contains two global information: the customer's id and the login date. To reflect such information in the user access graph, graph *P* associates two global data: *g_cid* and *g_date*. The value of *g_date* usually will not changed after a user login, thus it is a *global constant* and *g_cid* is a *global variable*. Graph *P3* inherits the global data of P, and it has an additional information/data: treatment_id (g_tid). However, global variable g_cid in *P3* becomes a global constant since the value will never be changed in *P3*.

The value of global variables may change when an event is submitted. For example, we don't know the value of treatment id until *q33* is submitted. When *q33* is submitted, the treatment id (g_tid) will be the value embedded in *q33*. Thus, we assign a local variable *l_tid* as the parameter of *v33*, and *v33* associates with an action: *g_cid=l_cid*. The parameter of *v34* is global variable *g_tid*.

Definition: database user session - a database user session consists of a sequence of queries which aim to finish certain function or task.

Definition: user access graph - we use user access graphs to describe the query execution orders within a user session and the orders between user sessions. A user access graph is a directed graph $G_p(V_p, E_p)$ which associates with a set of global variable $F_p = (f_1, ..., f_n)$ and a support value τ_p. $V_p = (v_1, ..., v_m)$ is a set of user access events or user access graphs. The parameter of a node is either a constant, a global variable, or a local variable. A directed edge $e_k = (v_i, v_j, \sigma_{v_i \to v_j})$ means that v_j follows v_i with a probability of $\sigma_{v_i \to v_j}$. Some nodes associate with actions that change the value of global variables. G_p has one start node and one or more end nodes.

Definition: basic user access graph - a basic user access graph is a user access graph whose nodes are user access events.

Table 1. An instance of treatment-history-retrieving procedure.

Label	Query	Business meaning
q30	select count(t_id) from treatment where c_id = 'c101'	get treatment count.
q31	select t_id from treatment where c_id = 'c101'	get treatment id
q32	select t_date from treatment where customer_id = 'c101'	get treatment date.
q33	select * from treatment_details where treatment_id = 't202'	get treatment details.
q34	select * from treatment_payment where treatment_id = 't202'	get payment information.

From Table 1, we observe that query $q34$ can not be anticipated until query $q33$ is submitted. Thus, the corresponding user access event $v34$ is determined by $v33$. We call events whose parameters are either constants, global constant or global variables *determined events* since the corresponding queries can be known before they are submitted. Other events are called *undetermined events*. For example, event *v33* is undetermined event since it has a local variable *l_tid*, and the others are determined events. A determined event can be *result-determined* by the results of other events, *parameter-determined* by the parameters of other events, or it only depends on constants or global constant which is called a *graph-determined* event. A graph-determined event does not depend on any other events. The difference between an undetermined event and a result-determined event is that the parameters of a result-determined event can be derived from query result. For example, *v31*, *v32* are graph-determined event, and *v34* is parameter-determined by *v33*.

Definition: determined/undetermined event - an event v_i is a determined event in a graph *p*, if and only if the parameters are either constants or global variables. Otherwise, v_i is undetermined.

Definition: graph/result/parameter determined event - 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. Event v_j is parameter-determined by v_i, if and only if one or more parameters are unknown until v_i is submitted.

There are many user access graphs in a client application or a group of users. We are interested in frequent user access graphs, i.e., the graph support is bigger than a predefined threshold τ. The **user access patterns** of a client application or a user group contains:

- A collection of frequently-performed queries;
- A collection of time-consuming or resource-consuming queries;
- A collection of frequent user access event associated with parameter distribution;
- A collection of frequent user access graphs which have a larger support than a predefined threshold τ.

3 Usages of User Access Patterns

User access patterns can be obtained by applying various data mining and statistic analysis methods on the database workloads or traces. Mining user access patterns usually involves the following steps: query clustering and classification; user and session identification; session clustering and classification and session modelling (i.e., generating user access graphs). In this paper, we will concentrate on using user access patterns to improve system performance, and assume the patterns are already obtained. We summarize the usage of user access patterns as follows:

The queries submitted by a database user are logically correct, but may not be executed efficiently. Database users can re-design the submitted queries according to the user access patterns. From the user access patterns, we can find the bottleneck of the system, i.e., the resource-consumed queries and the frequently-performed queries. Efficiently rewriting these queries will improve the system performance. Since each user access event represent a set of SQL queries, we can redesign the corresponding SQL template which may result in a great improvement on the overall system performance.

We can also reconstruct the query execution orders, i.e., redesign the user access graphs. In Section 4.3, we will show that this approach can also achieve a better performance.

User access patterns can help to tune the database system. There are many studies done on tuning DBMS through analyzing database workloads, such as index tuning [4] and materialized view suggesting [1, 5]. These techniques may provide useful suggestions for improving system performance, but the drawback is that they do not distinguish the step of finding user behaviors with the step of using the behaviors for query optimization. In our approach, we assume user access patterns are already mined from representative workloads. Since user access patterns represent the user behaviors, tuning the database system based on the user access patterns is better than analyzing the queries in a workload.

User access patterns can help to anticipate and prefetch queries. The submitted queries usually follow the user access graphs. Thus by matching a user request sequence with the user access graphs, we can predict future queries. Pre-computing the answers of future queries can reduce the query response time. Some extra step is needed to predict the corresponding queries. For example, we can not predict the corresponding query of an un-determined event since some local variable is unknown until the query is submitted. When a query is submitted, all queries that are parameter-determined by it are known, and can be predicted. And when a query finishes execution, all queries that are result-determined by it are known, and can be predicted.

User access patterns can also be used for query caching. We can store the answers of time-consumed queries or frequently-performed query to avoid repeated computing. In some cases, semantic relationships exist between the queries submitted by one user. Such relationships can help to rewrite and cache a query to answer multiple queries. For example, the predicate of query $q31$ and $q32$ in Table 1 are the same. Thus we can submit an alternative query $q31'$ – *"select count(*), t_id,t_date from treatment where customer_id = 'c101'"* – to answer both queries. In [18], we propose three types of solutions to rewrite the queries in a basic 'user access path. Given two consecutive queries *u* and *v*, a *sequential-execution* (*SEQ*) solution prefetches the answer of *v* which *u* is submitted. We may also submit the union query $(u \cup v)$ to answer both *u* and *v* , and it is called the *union-execution (UNI)* solution. The third solution is called *probe-remainder-execution (PR)* solution. In this solution, an extended version of query *u*, referred to as u', is submitted and cached. It 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 that are not in the cache. 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.

4 Experimental Results

4.1 SQL-Relay: an Event-driven, Rule-based Database Gateway

Since the submitted queries have certain format and follow certain order, it is reasonable to define certain rules to guide the execution of these queries. Thus, we proposed a new database gateway, SQL-Relay [17]. Unlike other database gateways and database caching servers, SQL-Relay treat each query as one type of event, and use pre-defined query execution rules to process it. It also traces user request sequences for query prediction. Database user can pre-define query execution rules to guide the execution of the events. There are three types of execution rules: global rewriting rules, local rewriting rules and prefetching rules. A global rewriting rule aims to rewrite a single query and improve it's execution performance, i.e., generating an equivalent query which can be execution more efficiently. A prefetching rule pre-fetches the answer of future queries based on the current request sequence. When semantic relationships exist between the queries of a user access path, a local rewriting rule can make use of such semantic relationship to rewrite the current query to answer multiple queries. The query execution rules can be obtained by analyzing user access patterns. *SQL-Relay* also include several functions to manipulate the query answers, such as *projection*, *sum*, *count* and simple selections. *SQL-Relay* is also a query caching server. It manages two kinds of caches, the global cache and the local cache. The global cache candidate can be chosen from the time-consumed queries and frequently-performed queries in the user access patterns. Query answers generated by applying prefetching rules or local rewriting rules are stored as local caches. We implement *SQL-Relay* by using Java language, and in the next section, we will show the experimental results by using *SQL-Relay* in an OLTP application.

4.2 Experimental Results for an OLTP application

Now, we present the experimental result of an OLTP application. A clinic office has a client application which helps the employee check in patients, make appointments and sell products. After preprocessing the database workload collected in 8 hour observation time , we found 12 instances of the application. The query log has 9,344 SQL statements and 190 SQL templates, where 72% (136) of queries are *SPJ* (select-project-join) queries. Few tables (6 out of 126) are frequently updated, and very few records are affected by a modification query (insertion, deletion or update). Thus, the cache maintenance cost is not too high.

718 sequences are found from the log. These sequences can be classified into 21 basic user access graphs with a *support* value bigger than 10. We require each node or path must have a minimal support value 3 or a confidence value 0.1. Representative user access graphs are shown in Figure 2. User access graph *P1_1*, *P1_2* and *P1_3* are high-level graphs. An instance of user access path *P1* which retrieves a given customer's profile is listed in Table 2. *P1* has four global parameters: user id (*g_uid*), customer id (*g_cid*), branch id (*g_bid*), and login date (*g_date*), where *g_date* is a constant global variable. Event *v30,v9,v47* are un-determined event, and event *v10* and *v20* are parameter-determined by *v9*. We can submit a query *"select * from customer where cust_num='1074'* to answer both query *q9* and *q20*, and partly answer query *q10*. Therefore, we can define a local rewriting rule for *P1*.

From the user access patterns, we generate 19 query prefetching rules and 7 local rewriting rules and deploy them on *SQL-Relay*. The global cache candidates are the selection query whose result size is smaller than *10K*, and is managed by using *LRU* replacement policy. Whenever the content of a base table changes, the corresponding

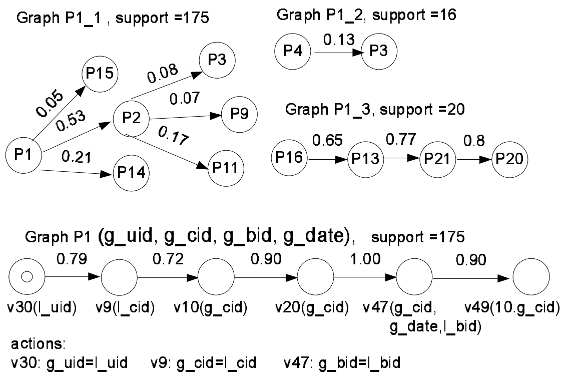


Fig. 2. User access graphs.

caches based on that table are discarded. We use *MySQL 4.0*[1] as our database server. It features a server-side *query cache* function, and we compare its performance with that of the *SQL-Relay*. The connection speed between the database server 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 support values of the user access graphs, and compare the cache performance under the following conditions: (1) executing queries without cache, (2) executing queries with 128K server cache, (3) only using query prefetching rules on *SQL-Relay*, and (4) using local rewriting rules and query prefetching rules together. The query response time is calculated at the client side. It includes server processing time, network transmission time, cache processing and maintaining time. Our result show that the response time of case 4 is the shortest since both network transmission time and server processing time are reduced. The prefetching case has a shorter response time than the server caching, but it doesn't improve server performance and has heavier network traffic than the later.

4.3 Experiment Results for TPC-W Benchmark

TPC Benchmark^{TW} (TPC-W) [15] is a transactional web benchmark that simulates the activities of a business oriented transactional database-driven web server. TPC-W models an on-line bookstore that consists of 14 browser/web server interactions, and each interaction consists of a sequence of SQL queries. TPC-W simulates three different profiles/users by varying the ratio of browse to buy: primarily shopping (WIPS), browsing (WIPSb) and web-based ordering (WIPSo). The benchmark also specifies the frequency of the web interactions and the transition probabilities matrix. Thus, we can derive user access patterns from the specification.

In [2], a TPC-W Java implementation is proposed. The remote browser emulation (RBE), web merchant application logic are implemented completely in Java language. We adopt the implementation to *Tomcat* application server, and *Microsoft SQL Server 2000* database. In the original implementation, the queries in a web interaction do not

[1] MySQL is registered trademark of the MySQL AB (http://www.mysql.com).

Table 2. An instance of user access path P1

ID	Statement
q30	select authority from *employee* where employee_id =**'1025'**
q9	select count(*) as num from *customer* where cust_num = **'1074'**
q10	select card_name from customer t1,member_card t2 where 1.cust_num = **'1074'** and t1.card_id = t2.card_id
q20	select contact_last,contact_first from *customer* where cust_num = **'1074'**
q47	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'**
q49	select top **10** contract_no from *treatment_schedule* where cust_id = **'1074'** order by checkin_date desc

share common connections. We modify the implementation to make all queries in a web interaction share the same connection. We collect TPC-W database workload which contains the mix of 15 concurrent RBEs over an 8 hour's observation time. The workload contains 25,994 SQL queries. There are only 46 SQL templates in the workload, thus well-design SQL-templates can achieve a better system performance. The average disk I/O for the queries of Best Seller web interaction is 4,260 block, and the average I/O for all other queries is only 170 block. Thus, they are the most resource-consumed queries. We also observe that the queries of *Best Sellers*, *New Products* and *Subject Search* web interaction are the most frequently-performed queries. Thus, caching the result of these queries can obtain a great performance improvement. Other studies also support our observation [16, 10].

User access patterns can also help us to design the queries. For example, table 3 lists the pseudo code for the *order display* web interaction. It contains 5 consecutive queries, where $\sigma_{q1\rightarrow q2}$ is the frequency of the valid users, and $\sigma_{q2\rightarrow q3}$ is the frequency of the users who have at least one order. The think-time between these queries is very short, thus prefetching rules are not suitable. We still can rewrite these queries to achieve better performance. Query *q2,q3,q4* are result-determined by their predecessor, and query *q5* is result-determined by *q2*. We can split join query q3 into query q3_1: *select * from customer where cid=@c_id* and query q3_2: *"select orders.*, address.* country.* from customer,address,country,orders where o_id=@o_id and o_bill_addr=addr_id and addr_co_id=co_id*. Since both column *c_uname* and *c_id* are the key of table *customer*, thus query q3_1 is equivalent to query *q1': select * from customer where c_uname=@c_uname and c_passwd=@c_passwd* has containing-match relationship with *q1*. We can replace *q1* with *q1'*, and it is called *rewriting case1*.

In the second case, we try to rewrite query *q3* and *q4*. Since o_id is the primary key, and o_ship_addr is the foreign key, we can join *q3* and *q4* together as *q3'*. The advantage is that we make use of the foreign key constraints, and only submit one query to the server. The disadvantage is that we introduce a new join condition. In the third case (case 3), we first rewrite query *q3* as *q3_1* and *q3_2*, then replace query *q1* with *q1'* and join *q3_2* and *q4* as well. In case 4, we consider combining query *q2*, *q3*,

Table 3. Pseudo code for the order display web interaction

ID	Statement
q1	select @c_cid = c_id from customer where c_uname=@c_uname and c_passwd=@c_passwd
q2	select @o_id = max(o_id)from orders where o_c_id=@c_id
q3	select customer.*, orders.*, address.* country.* from customer,address,country,orders where o_id=@o_id and c_id=@c_id and o_bill_addr=addr_id and addr_co_id=co_id; set @a_id = result.o_addr_id
q4	select address.*, country.* from address,country where addr_id=@a_id and addr_co_id=co_id
q5	select * from order_line,item where ol_i_id=i_id and ol_o_id=@o_id

and *q4* together. Since *q2* will return a single value which is only used by query *q3*, we can rewrite *q2* as the sub-query of the join of *q3* and *q4*. In the last case (case 5), we first rewrite query *q3* as *q3_1* and *q3_2*, then replace query *q1* with *q1'* and rewrite *q2* and *q3_2* and *q4* together.

The performance of these rewriting solutions depends on the internal query process mechanism of DBMS. We create necessary indices according to the definition of *TPC-W* benchmark, and test the system performance under two different database servers: *Microsoft SQL Server 2000* [2] and *MySQL 4.0*. The benchmark does not specify the frequency of the valid users and the frequency of the users who have at least one order, we set them as *0.95* and *0.80* respectively. The client application is implemented by using *Java* language which has a connection speed of *56K bps* with the server. Table 4 illustrates the experimental results, and case *0* corresponds to the original queries. The two servers have different results since they have different query processing mechanism. For *SQL Server 2000*, case *4* and *5* generate fewer requests to the server and have faster response time than the other cases. However, they have more server *I/O*. Meanwhile, case *2* and *3* have a shorter response time than case *0*, and they have the same amount of server *I/O* as case *0*. Case *1* has no significant improvement over case *0*. *MySQL* server does not support sub-query, thus case *4* and *5* are not applicable. Case *2* and *3* have shorter response time than case *0* and *1*. And case *1* and *3* have less server *I/O* than case 0 and *2*, while case *1* has the lowest network traffic [3].

5 Related Work

A survey of the techniques and methodologies for the construction of workload models in different application domains is presented in [3]. Some previous workload studies focused on describing the statistical summaries and run-time behavior [14, 20, 8], clustering database transactions[21, 19, 11], predicting the buffer hit ratio [6, 7], and improving caching performance [13, 12, 18]. Chaudhuri *et al* [14] suggest to use SQL-like primitives for workload summarization, a few examples of workload summarization and the possible extensions to SQL and the query engine to support these primitives are

[2] SQL Server 2000 is registered trademark of the Microsoft Corporation.

[3] The experimental results presented in the paper should not be used for the benchmark performance comparison between the two database servers

Table 4. Performance for Order Display Interaction (per 100 instances)

		SQL Server 2000			MySQL 4.0		
	queries per instance	response time (s)	server I/O (pages)	sent/received (packets)	response time (s)	server I/O (pages)	sent/received (Kbytes)
case0	5	518	3557	1810/1810	259	2259	611/ 53
case1	5	501	3576	1810/1810	206	1965	596/ 49
case2	4	464	3557	1510/1510	190	2259	567/ 90
case3	4	470	3556	1510/1510	189	1965	571/ 86
case4	3	401	3648	1210/1210	N/A	N/A	N/A
case5	3	407	3721	1210/1210	N/A	N/A	N/A

also discuss in the paper. n [20], a relational database workload analyzer (REDWAR) is developed to characterize the workload in a DB2 environment. The study focuses on statistical summaries, such as averages, variations, correlations and distributions, and description of the run time behavior of the workload. In [8], Hsu *et al* analyze the characteristics of the standard workload TPC-C and TPC-D, and examine the characteristics of the production database workloads of ten of the world's largest corporations. Yu *et al* [19] propose an affinity clustering algorithm which partitions the transactions into clusters according to their database reference patterns. Nikolaou *et al* [11] introduce several clustering approaches by which the workload can be partitioned into classes consisting of units of work exhibiting similar characteristics. Dan *et al* [6, 7] analyze the buffer hit probability based on the characterization of low-level database access (physical pages). They make distinctions among three types of access patterns: locality within a transaction, random access by transactions, and sequential accesses by long queries. Sapia [13, 12] discusses the PROMISE (Predicting User Behavior in Multidimensional Information System Environment) approach which provides the cache manager with transaction level access patterns to make prediction for OLAP system. In [18], we propose algorithms to analyze the semantic relationship between the events of a user access graph, rewrite and cache the current query based on the semantic relationship between it and future queries.

6 Conclusion

In this paper, we use the user access events to represent the static features of a database workload, and use the user access graphs to represent the dynamic features. To our knowledge, this is the first attempt to analyze database user's access patterns systemically and completely. The experimental results presented in this paper show that the application of user access patterns can improve system performance greatly. Database vendors can also use the idea of *SQL-Relay* proposed in the paper to develop intelligent database gateway.

References

1. Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB Conference*, pages 496–505, 2000.
2. Todd Bezenek, Harold Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles, and Mikko Lipasti. Characterizing a Java implementation of TPC-W. In *Third CAECW Workshop*, 2000.
3. Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8):1136–1150, 1993.
4. Surajit Chaudhuri and Vivek R. Narasayya. Microsoft Index Tuning Wizard for SQL Server 7.0. In *SIGMOD Conference*, pages 553–554, 1998.
5. Surajit Chaudhuri and Vivek R. Narasayya. Automating statistics management for query optimizers. In *ICDE Conference*, pages 339–348, 2000.
6. Asit Dan, Philip S. Yu, and Jen-Yao Chung. Database access characterization for buffer hit prediction. In *Proc. of the Ninth ICDE, 1993*, pages 134–143. IEEE Computer Society, 1993.
7. Asit Dan, Philip S. Yu, and Jen-Yao Chung. Characterization of database access pattern for analytic prediction of buffer hit probability. *VLDB Journal*, 4(1):127–154, 1995.
8. W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*, 40(3), 2001.
9. O. Klaassen. Modeling data base reference behavior. In *Computer Performance Evaluation: Modelling Techniques and Tools*, page 47. NorthHolland, 1992.
10. Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of VLDB 2001*, pages 191–200, 2001.
11. Christos Nikolaou, Alexandros Labrinidis, Volker Bohn, Donald Ferguson, Michalis Artavanis, Christos Kloukinas, and Manolis Marazakis. The impact of workload clustering on transaction routing. Technical Report TR98-0238, 1998.
12. C. Sapia. PROMISE: modeling and predicting user query behavior in online analytical processing environments. FR-2000-001, Forwisee Technical Report, 2000.
13. Carsten Sapia. PROMISE: predicting query behavior to enable predictive caching strategies for OLAP systems. In *DAWAK*, pages 224–233, 2000.
14. Vivek R. Narasayya Surajit Chaudhuri, Prasanna Ganesan. Primitives for workload summarization and implications for sql. In *VLDB 2003, Berlin, Germany*, pages 730–741. Morgan Kaufmann, 2003.
15. Transaction Processing Performance Council (TPC). TPC Benchmark-W (Web Commerce) - standard specification revision 1.6, Feb 2002.
16. Wayne D. Smith, Intel Corporation. TPC-W: Benchmarking an ecommerce solution, Feb 2000.
17. Qingsong Yao and Aijun An. SQL-Relay: An event-driven rule-based database (demonstration). In *International Conference on Web-Age Information Management*, 2003.
18. Qingsong Yao and Aijun An. Using user access patterns for semantic query caching. In *Database and Expert Systems Applications*, 2003.
19. P. S. Yu and A. Dan. Performance analysis of affinity clustering on transaction processing coupling architecture. *IEEE TKDE*, 6(5):764–786, 1994.
20. Philip S. Yu, Ming-Syan Chen, Hans-Ulrich Heiss, and Sukho Lee. On workload characterization of relational database environments. *Software Engineering*, 18(4):347–355, 1992.
21. Philip S. Yu and Asit Dan. Impact of workload partitionability on the performance of coupling architectures for transaction processing. In *Proc. of the Fourth IEEE SPDP Conference*, pages 40–49, 1992.