Mining and Modeling Database User Access Patterns

Qingsong Yao, Aijun An, and Xiangji Huang*

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

Abstract. We present our approach to mining and modeling the behavior of database users. In particular, we propose graphic models to capture the database user's dynamic behavior and focus on applying data mining techniques to the problem of mining and modeling database user behaviors from database trace logs. The experimental results show that our approach can discover and model user behaviors successfully.

1 Introduction

Workload analysis has played an important role in optimizing the performance of database systems. While most work on database workload analysis focuses on providing statistical summaries and run-time behavior on the physical resource level of the database system, it has been brought into attention that analysis of task-oriented user sessions provides useful insight into the query behavior of the database users [7,8]. A session is a sequence of queries issued by a user (or an application) to achieve a certain task. It consists of one or more database transactions, which are in turn a sequence of operations performed as a logical unit of work. Analysis of sessions allows us to discover high-level patterns that stem from the structure of the task the user is solving.

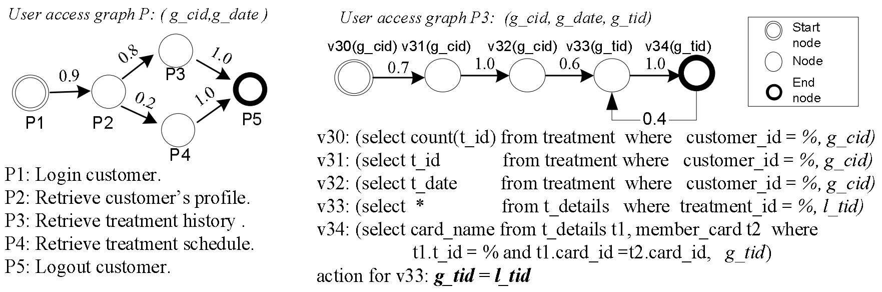
In this paper, we first describe our approach to modeling database user behaviors. We use the concept of *user access event* and *user access graph* to represent the database queries and the dynamic relationship among the queries. Then, we describe how we apply data mining techniques to mine database users' access patterns from database traces. Our contributions in this paper are summarized as follows. First, we present a complete process for mining user access patterns from database trace logs, including data preprocessing, session identification, session clustering and the generation of user access graphs. Second, we present the use of Markov models to build the user access graphs, and provide experimental results showing that the discovered user access graphs can model the database user behavior well.

2 User Access Graph

The SQL queries submitted by a database user are not random. They follow business rules or logics. We use *user access event* and *user access graph* to represent the query

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

© Springer-Verlag Berlin Heidelberg 2006


User access graph P: (g_cid,g_date)
P1
P2
P3
P4
P5
0.9
0.8
0.2
1.0
1.0
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
0.4
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 = % and t1.card_id =t2.card_id, *g_tid*)
action for v33: ***g_tid = l_tid***

Fig. 1. An example of user access graphs

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

ID	Query	Business meaning
q30	select count(t_id) from treatment where customer_id = 'c101'	get customer c101's treatment count
q31	select t_id from treatment where customer_id = 'c101'	get a list of treatment id
q32	select t_date from treatment where customer_id = 'c101'	get a list of treatment date.
q33	select * from treatment_details where treatment_id = 't202'	get the details of one treatment
q34	select * from treatment_payment where treatment_id = 't202'	get the payment information of one treatment

format and the query execution order, respectively. Given an SQL query, we 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 are only different in data values. In this paper, we use a *user access event* to represent queries with similar format. A user access event contains an SQL template and a set of parameters. For example, the SQL query *"select name from customer where id = 'c101' "* can be represented by event *("select name from customer where id = '%' ", g_cid)*, where *g_cid* is a parameter.

Given a set of SQL query sequences that have similar formats and execution orders, we use a *user access graph* to represent the query execution order. A user access graph is a directed graph that has one start node and one or more end nodes. Each node in the graph is a user access event or a user access graph. 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 $e_k : (v_i, v_j, \sigma_{v_i \rightarrow v_j})$, where $\sigma_{v_i \rightarrow v_j}$ is the probability of v_j following v_i, which is called the confidence of the edge. 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. For example, in a client application, there is a patient-information model that helps an employee to retrieve a patient's treatment history and treatment schedule. When a user logs in, the corresponding profile is retrieved. Then he/she can either retrieve the treatment history information or treatment schedule. Figure 1 shows two user access graphs for the patient-information model. Graph *P*, illustrated on the left side, is a high-level user access graph that describes the execution orders of five sub-models. 80% of the users retrieve treatment history information (since $\sigma_{P2 \to P3} = 0.8$). Graph $P3$, illustrated on the right side, is a basic user access graph that represents the treatment-history retrieval sub-model. Table 1 shows an instance of the treatment-history sub-model that contains four consecutive SQL queries.

A parameter in an event can be a constant, a dependent variable or an independent variable. The value of a constant parameter cannot be changed by any of the events in the user access graph. An independent variable can take different values and its value does not depend on any other variable or constant in the graph. The value of a dependent variable can be changed by an event and its value depends on some other variables in the graph. For example, g_cid is a constant parameter in the user access graph $P3$ on the right side of Figure 1, since its value does not change by any of the events on the graph. g_tid in event $v34$ is a dependent variable because its value is set by event $v33$ to be the value of parameter l_tid in $v33$, which is an independent variable. 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 not independent variables *determined events* since the corresponding queries are known before they are submitted. Other events are called *undetermined events*. For example, event *v33* is undetermined event, and *v31*, *v32*, and *v34* are determined event. 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 the latter can be derived from query result. For example, *v31*, *v32* are graph-determined event, and *v34* is parameter-determined by *v33*.

3 Discovering User Access Graphs

Our objective is to mine user access graphs from database traces. The procedure of mining user access graphs includes the following steps. First, database traces are collected and preprocessed, in which noisy and irrelevant data are removed and the log entry is transformed to a meaningful format. Second, every SQL query in the traces is transformed into an *SQL template* and a set of *parameters*. We treat a data value in a given SQL query as a parameter, and the SQL template can be obtained by replacing each parameter in the query with a wildcard character *'%'*. A *user access event* is assigned to each SQL template to represent a set of *similar queries*. By replacing the SQL statements in the log entry with the corresponding user access events, we obtain sequences of user access events, referred to as *event sequences*. In the third step, we apply a session identification method to detect session boundaries from an event sequence which contains multiple sessions submitted sequentially [3,10]. After session instances are identified, we further group them into session classes using a clustering method described in [9]. Each session class contains a set of session instances. Finally, for each session class, a user access graph is built. In the next section, we describe the last step that builds a user access graph from a set of session instances of the same class.

4 Modeling Database User Sessions

A session class g contains a set of session instances, $\{s_1, s_2, ..., s_n\}$. Each session instance is a sequence of requests. The procedure of modeling session classes contains three steps. First, we ignore the relationships among parameters in the requests and consider the request execution order only. In this step, we use Markov models to build the structures of the basic user access graphs, i.e., the nodes and the edges of the graphs, where nodes represent user access events. Second, we find the relationships between the parameters in use access events and introduce variables/actions to the graphs. Finally, to build a high-level user access graph, we replace each session instance s_i with the corresponding session class, and obtain a collection of session class sequences, one for each user. The idea of building low-level user access graphs can be used to build high-level user access graphs.

4.1 User Access Graph Generation with Markov Model

Let $X_1, X_2, ...X_n, ...$ be a sequence of random variables taking values from a finite set of values $S = \{x_1, x_2, ..., x_n\}$. The random variables are said to form a k^{th}-order Markov chain model if

$$P(X_i|X_1, X_2, ..., X_{i-1}) = P(X_i|X_{i-k}, ..., X_{i-2}, X_{i-1}), \qquad (1)$$

for all values of i. In other words, the current variable at time i depends on previous k variables. We can think of the values of X_i as *states*, and the Markov model as a finite state process with transitions between states specified by probabilities $P(x_i|x_{i-k}, ..., x_{i-2}, x_{i-1})$. The probabilities can be represented by an $n^k \times n$ matrix. k is called the order of the model. A k^{th}-order Markov model can always be converted into an equivalent first-order Markov model. We can introduce random variables as $Z_i \doteq X_{i-k+1}, X_{i-k+2}, ...X_i$, and the Z process forms a first-order Markov model [4].

A zero-order Markov model makes prediction based on the action reference statistics, and a first-order Markov model makes predictions based on the last action performed by the user. In general, zero-order Markov model and first-order Markov model have limited success in representing the dynamic user behavior since these models do not look far into the past. A k^{th}-order Markov model make predictions by looking at the last k actions performed by the user, which lead to a state-space that contains all possible sequences of k actions. The large number of possible states leads to a large transition probability matrix.

To build a user access graph for each session class, we generate a higher-order Markov model[1] from a group of sessions. We treat each state of the Markov model as a node in the user access graph, and each state transition corresponds to an edge in the graph. Thus, the Markov state transition diagrams describe the structure of a user access graph. The procedure for building a Markov model is straightforward. We first scan

[1] Strictly speaking, the model used is first-order Markov models, where every state is a sequence of k requests. But since it is transformed from a k^{th}-order Markov in which every request corresponds to a state, it is equivalent to a higher-order Markov model.

all session instances once, generate the possible states and the state transition probabilities. Once a session instance $s =< r_1, ..., r_n >$ is retrieved, we add a special request *"start"* at the beginning of s, and s forms totally *n+1* states: $(start)$, $(start, r_1)$,..., $(r_{n-k}, ..., r_n)$. The state frequency and state transition frequency are updated correspondingly. In the second step, the states and state transitions are pruned according to the state support threshold (i.e., the minimal support value for the node) and the confidence threshold (i.e., the minimal confidence value for the edge). We refer to such models as the *pruned Markov* models. A corresponding state transition diagram can also be easily constructed from the transition matrix. By using pruned Markov models, the number of states and the number of state transitions are reduced.

First Order Markov Model

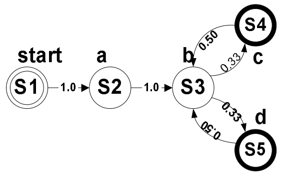
S1: (start), S2: (a), S3: (b),
S4: (c), S5: (d),

	Freq	S1	S2	S3	S4	S5
S1	4	0	4	0	0	0
S2	4	0	0	4	0	0
S3	6	0	0	0	2	2
S4	2	0	0	1	0	0
S5	2	0	0	1	0	0

Second Order Markov Model

S1: (start), S2: (start, a), S3: (a,b), S4: (b,c)
S5: (b,d), S6: (c,b), S7:(d,b),

	Freq	S1	S2	S3	S4	S5	S6	S7
S1	4	0	4	0	0	0	0	0
S2	4	0	0	4	0	0	0	0
S3	4	0	0	0	2	2	0	0
S4	2	0	0	0	0	0	1	0
S5	2	0	0	0	0	0	0	1
S6	1	0	0	0	0	0	0	0
S7	1	0	0	0	0	0	0	0



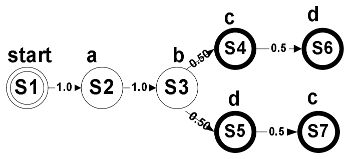


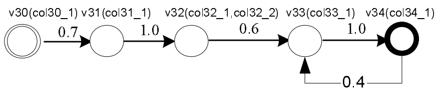
Fig. 2. First-order and second-order pruned-Markov Model

For example, given four session instances $< a, b, c >$, $< a, b, c, b >$, $< a, b, d >$, $< a, b, d, b >$, the first-order and second-order pruned-Markov models and their corresponding transition diagrams are illustrated in Figure 2. The confidence threshold is 0.33 and the support count threshold is 1. We observe that first-order model has fewer states than the second-order model, but it cannot distinguish the two requests of b in session $< a, b, d, b >$ and $< a, b, c, b >$. We can also observe that the number of states in a pruned second-order models is not large (compared with $4^3 = 64$ in the un-pruned model), and it can model the dynamic behavior of the session class well.

4.2 Finding Relationship Between Nodes

The relationship discovery problem is defined as follows. Given a user access graph G_s and a set s of sessions $\{s_1, ..., s_n\}$ belonging to it, find the relationships between the node parameters in G_s, and introduce constants, variables and edge actions into G according to the relationships. In this step, each request in a session is an SQL query and can be represented by using an user access event and a set of values. Each data value corresponds to a parameter of the event. Meanwhile, each request can be mapped into a node in the user access graph. The relationships between node parameters can be discovered by analyzing the data values between the requests.

We assume that all parameters in the nodes of G make a virtual relation R, where each parameter is one attribute of R. Each session s_i, which contains a sequence of queries, corresponds to a tuple of R. The data values in all the queries in s_i are the attribute values of that tuple. Therefore, the set s of sessions make an instance of relation R, referred to as r. Thus, the problem of finding parameter relationships of a user access graph G_s with sessions s becomes the problem of finding functional dependencies and dependency functions in an instance r of relation R.



v30: (select count(t_id) from treatment where c_id = *%, col30_1)*
v31: (select t_id from treatment where c_id = *%, col31_1)*
v32: (select t_id from treatment where c_id = *%* and *status=%, col32_1, col32_2)*
v33: (select * from treatment_details where t_id = *%, col33_1)*
v34: (select * from payment_details where t_id = %, *col34_1*)

Fig. 3. Original user access graphs

Suppose a user access graph listed in Figure 3 is obtained. The graph contains five user access events where each event contains a set of parameters. we first assign a unique name to each parameter. The corresponding virtual relation contains six attributes, referred to as *col30_1, co31_1, col32_1, col32_2, col33_1, col34_1*, respectively. Table 1 shows a session that contains five consecutive queries. The corresponding tuple of R is illustrated in Table 2, which is obtained by extracting the data values from the queries, and mapping them to the corresponding attributes.

Table 2. A record of relation R

col30_1	**col31_1**	**col32_1**	**col32_2**	**col33_1**	**col34_1**
’c101’	’c101’	’c101’	’1’	’t202’	’t202’

From the virtual relation instance, we can find the functional dependencies. The parameter relationship discovery procedure usually contains the following steps:

1. Virtual relation and relation instance generation. In this step, virtual relation R and relation instance r are generated from sessions s and user access graph G_s.
2. Functional dependency inference. In this step, functional dependencies are inferred from r by using a functional dependency inference algorithm. Two kinds of special dependencies, $\{\rightarrow Y\}$, and $\{X \leftrightarrow Y\}$, are two common cases in our inference problem, which can be found in an efficient way. Dependency $\{\rightarrow Y\}$ means that the value of Y is a constant and will not change. It can be discovered by examining the cardinality of Y, which is the number of distinct values of attribute Y in the relational instance. Dependency $\{X \leftrightarrow Y\}$ means that X and Y agree on all possible values.

3. Dependency function discovery. In this step, dependency functions are discovered from relation instance *r*. A dependency function, for a given functional dependency $X \rightarrow Y$, can help to obtain a value of Y by giving the value of X. When X and Y take numeric values, we use regression analysis to find a function. When the target attribute Y is categorical, we can treat the relation instance *r* as the training data, and each tuple in *r* has a class label *y* that is the corresponding value of attribute *Y*. Thus, a set of classification rules can be obtained from the training data. The classification rules can be used to describe the relationship among query parameters, and can be viewed as the dependency functions.
4. Graph variable and edge action introduction. In this step, graph variables and edge actions are introduced according to the dependencies and functions discovered in step 2 and 3.

5 Experimental Results

We evaluated our modeling method on a clinic OLTP application. The clinic is a private physiotherapy clinic located in Toronto. It has five branches across the city. It provides services such as joint and spinal manipulation and mobilization, post-operative rehabilitation, personal exercise programs and exercise classes, massage and acupuncture. The client program consists of several models or components. Each model consists of sequences of SQL statements, and the query execution order is controlled by the business logics embedded in the model. The execution of these models in an instance of the client program also has certain rules. Each day, the client applications installed in the branches make connections to the center database server, which is Microsoft SQL Server 7.0. In each connection, a user may perform one or more tasks, such as checking in patients, making appointments, displaying treatment schedules, explaining treatment procedures and selling products. The trace file is collected by using Microsoft SQL Profiler [2]. The SQL Profiler can monitor all events that occur in the SQL Server database, and a user can define the events to be monitored manually. We use the SQL Profiler to collect all SQL queries submitted by the client application within a period of observation time. The database trace log (400M bytes) contains *81,417* events belonging to 9 different applications,such as front-end sales, daily report, monthly report, data backup, and system administration. The target application of the paper is the front-end sales application. After preprocessing the trace log, we obtain 7,244 SQL queries, 18 database connection instances of the front-end sales application. 2989 types of queries are found from the trace log, and only 4% of the queries have a frequency over 5. The queries are classified into 190 user access events, which have an average frequency of 38. 18 user access event sequences are obtained by replacing the queries with the corresponding user access events, each sequence corresponds to one client application instance.

We choose half of the database traces as the training data, and randomly select four user access event sequences from the remaining traces as the test data. Session identification is first performed on the data sets and then the sessions in the training data are clustered using our session clustering method into 21 session clusters (classes). We

[2] SQL Profiler is registered trademark of Microsoft Corporation.

prune out the clusters that have less than 10 session instances. We construct the corresponding user access graphs for the remaining session clusters by using the algorithm proposed in Section 4. In the experiment, we use the minimal support value of 3 and the minimal confidence value of 0.05. Representative user access graphs are shown in Figure 4. User access graph *P1_1*, *P1_2* and *P1_3* are high-level graphs. An instance of user access graph *P1* which retrieves a given customer's profile is listed in Table 3. The graph instance can be interpreted as employee *'1025'* retrieve customer *'1074'* 's profile and treatment schedule at branch *'scar'* on *2003-03-04*. Thus, the graph has four parameters: user id (*g_uid*), customer id (*g_cid*), branch id (*g_bid*), and login date (*g_date*), where *g_date* is a constant. We observe that once query q9 is submitted, the format of query q10, q20 and q49 are determined, thus they can be predicted. In the correspond graph *P1*, events *v30,v9,v47* are un-determined event, and events *v10*, *v20* and *v49* are parameter-determined by *v9*.

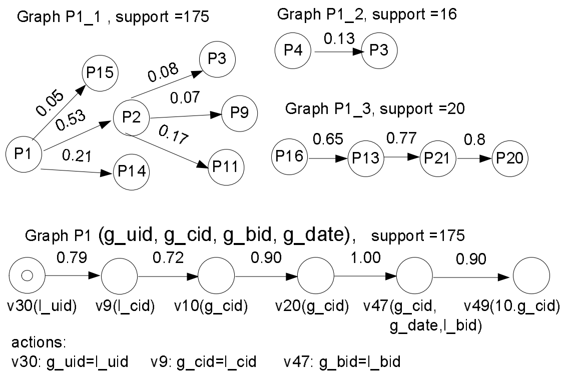


Fig. 4. User access graphs

Table 3. 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 = **'2003/03/04'** and t1.branch = **'scar'**
q49	select top **10** contract_no from *treatment_schedule* where cust_id = **'1074'** order by checkin_date desc

To evaluate the user access graphs, we use them to predict the next request from a sequence of the request history $x : (x_1, x_2, ..., x_n)$, where x is part of a session instance in the test data. The method for predicting next request is straightforward. for each session class g_i, the probability that a request y_i be the next request under g_i is:

$$P(y_i|x, g_i) = P(y_i|x_1 x_2 ... x_n, g_i) = P(y_i|x_{n-k+1} ... x_n, g_i)$$

where $P(y_i|x_{n-k+1}...x_n, g_i)$ can be obtained from the transition probabilities of g_i. We predict next request to be y_i only when $P(y_i|x, g_i)$ exceeds a certain threshold and $P(y_i|x, g_i)$ achieves the maximum value among all classes. If the next request is predicted correctly, we said it is a "match". We use *F-Measure* to measure the prediction performance, which is defined as follows:

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall},$$

where *precision* is defined as the ratio of the number of "matches" to the total number of predicted requests and the *recall* is the hit-rate, which is the portion of the requests that are correctly "matched".

The result for request prediction is shown in Table 4, where there are 5,490 requests in the test data. The prediction threshold is 0.8. Table 4 shows that the overall *precision* of the prediction algorithm is very high, but the *recall* is not. The reason is that we can not make prediction when there is limit request histories, which usually occurs when a session just begins. The result shows that our method can effectively represent the users' access pattern and can predict users' next requests well. In this application, a 5-order Markov-model performs the best.

Table 4. Request Prediction Performance

Markov order	predicted requests	matched requests	precision	recall	F-Measure
1	2718	2627	0.97	0.48	0.64
2	2870	2776	0.97	0.51	0.67
3	2874	2791	0.97	0.51	0.67
4	2894	2811	0.97	0.52	0.67
5	2918	2839	0.97	0.52	0.68
6	2927	2848	0.97	0.52	0.68
7	2918	2842	0.97	0.52	0.68
8	2926	2850	0.97	0.52	0.68
9	2377	2345	0.99	0.43	0.60
10	2425	2393	0.99	0.44	0.61
11	1979	1947	0.98	0.36	0.52
12	1979	1947	0.98	0.36	0.52

6 Related Work

Previous workload studies focus on describing the statistical summaries of run-time behavior [7,12,2], clustering database transactions [11,5], predicting the buffer hit ratio [1], and improving caching performance [6,8]. Chaudhuri *et al* [7] 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 also discussed in the paper. In [12], a relational database workload analyzer (REDWAR) is developed to characterize the workload in a DB2 environment.

Their study focused on statistical summaries, such as averages, variations, correlations and distributions, and description of the runtime behavior of the workload. In [2], 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* [11] propose an affinity clustering algorithm which partitions the transactions into clusters according to their low-level database reference patterns. Nikolaou *et al* [5] introduce several clustering approaches by which the workload can be partitioned into classes consisting of units of work exhibiting similar characteristics. Dan *et al* [1] analyze the buffer hit probability based on the characterization of low-level database access to 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. However, all above mentioned clustering algorithms are based on the low-level reference patterns.

7 Conclusion

We have presented a new approach to mining and modeling database user access patterns. To our knowledge, this is the first attempt to analyze database user access patterns systematically. We use the user access events to represent the static features of a database workload, and use the user access graphs to describe database query execution orders. The mining results from our approach can be used to tune the database system and predict incoming queries based on the queries already submitted, which can be used to improve the database performance by effective query prefetching, query rewriting and cache replacement. Experiments show that our approach provides a promising avenue for mining and modeling database users' access behaviors. The work presented in the paper has a broader impact on the database and data mining fields. It can also be used on Web log analysis and DNA sequence analysis. We believe that our approach can help database vendors develop intelligent database tuning tools and rule-based database gateway. In the future, we plan to apply the proposed ideas to OLAP trace logs.

References

1. 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.
2. 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.
3. Xiangji Huang, Qingsong Yao, and Aijun An. Applying language modeling to session identification from database trace logs. *Knowledge and Information Systems: An International Journal (KAIS)*, 2006.
4. Frederick Jelinek. *Statistical Methods for Speech Recognition*. The MIT Press, 1998.
5. 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.
6. Carsten Sapia. PROMISE: Predicting query behavior to enable predictive caching strategies for OLAP systems. In *DAWAK*, pages 224–233, 2000.

7. Vivek R. Narasayya Surajit Chaudhuri, Prasanna Ganesan. Primitives for workload summarization and implications for sql. In *VLDB 2003, Berlin, Germany*, pages 730–741, 2003.
8. Qingsong Yao and Aijun An. Using user access patterns for semantic query caching. In *Database and Expert Systems Applications (DEXA)*, 2003.
9. Qingsong Yao, Aijun An, and Xiangji Huang. A distance-based algorithm for clustering database user sessions. In *ISMIS*, 2005.
10. Qingsong Yao, Xiangji Huang, and Aijun An. A machine learning approach to identifying database sessions using unlabeled data. In *DaWaK*, pages 254–264, 2005.
11. P. S. Yu and A. Dan. Performance analysis of affinity clustering on transaction processing coupling architecture. *IEEE TKDE*, 6(5):764–786, 1994.
12. 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.