Tsinghua Science and Technology

Volume 18 | Issue 6 Article 8

2013

Performance Prediction for Performance-Sensitive Queries Based on Algorithmic Complexity

Chihung Chi
School of Software, Tsinghua University, Beijing 100084, China

Ye Zhou
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Xiaojun Ye
School of Software, Tsinghua University, Beijing 100084, China

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 08/10 pp618-628
Volume 18, Number 6, December 2013

Performance Prediction for Performance-Sensitive Queries Based on Algorithmic Complexity

Chihung Chi*, Ye Zhou, and Xiaojun Ye

Abstract: Performance predictions for database queries allow service providers to determine what resources are needed to ensure their performance. Cost-based or rule-based approaches have been proposed to optimize database query execution plans. However, Virtual Machine (VM)-based database services have little or no sharing of resources or interactions between applications hosted on shared infrastructures. Neither providers nor users have the right combination of visibility/access/expertise to perform proper tuning and provisioning. This paper presents a performance prediction model for query execution time estimates based on the query complexity for various data sizes. The user query execution time is a combination of five basic operator complexities: $O(1)$, $O(\log(n))$, $O(n)$, $O(n\log(n))$, and $O(n^2)$. Moreover, tests indicate that not all queries are equally important for performance prediction. As such, this paper illustrates a performance-sensitive query locating process on three benchmarks: RUBiS, RUBBoS, and TPC-W. A key observation is that performance-sensitive queries are only a small proportion (20%) of the application query set. Evaluation of the performance model on the TPC-W benchmark shows that the query complexity in a real life scenario has an average prediction error rate of less than 10% which demonstrates the effectiveness of this predictive model.

Key words: query performance; data size; query complexity; performance-sensitive query

1 Introduction

Cloud computing is emerging today as a new paradigm for providing computing infrastructures and services over shared IT infrastructures using a pool of abstracted, virtualized, and scalable computing resources[1-3]. Along with cloud-based applications, Database as a Service (DaaS) aims to free Service Consumers (SC) from maintaining and managing complex IT infrastructures. One of the most significant issues for database services is the use of its resources and performance predictions for multi-tenant, multi-Database Management System (DBMS) tuning. Like all cloud computing systems, DaaS, which concurrently executes heterogeneous queries, needs to satisfy diverse application-level SLAs, especially high-level metrics such as service performance expectations. Thus, performance guarantees for DaaS are crucial for the infrastructure provider[4-7]. Much effort has been devoted to finding optimized query execution plans. However, Virtual Machine (VM)-based database services have little or no sharing of resources or interactions between applications hosted on shared infrastructures, so the providers and the users both do not have the right combination of visibility, access, and expertise to perform proper tuning and provisioning. Thus, cost-based or rule-based approaches used in traditional DBMS are not adequate for the database service performance

• Chihung Chi and Xiaojun Ye are with the School of Software, Tsinghua University, Beijing 100084, China. E-mail: chihungchi@gmail.com; yexj@tsinghua.edu.cn.
• Ye Zhou is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhouye09@mails.tsinghua.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2012-10-08; revised: 2013-05-13; accepted: 2013-06-08

optimization. Automated service optimization solutions are needed for the database service provided to solve DBMS tuning problems in multi-tenant environments. These problems require new query execution time performance predictive models which can predict the resource utilization and can estimate the achievable performance with a given set of resources.

The relationship between the query execution time and the database size can be described by the query complexity[8-10]. User queries can be decomposed into different types based on their time complexity as $O(1), O(\log(n)), O(n), O(n\log(n))$, and $O(n^2)$. Not all queries have equal effects on the database performance. Queries with complexity $O(1)$ or $O(\log(n))$ have little impact on the database performance and queries with complexity $O(n^2)$ seldom occur[11]. Finally, when the database volume varies rapidly, the DBMS needs only to focus on optimizing queries with complexity $O(n\log(n))$ and $O(n)$. In addition, not all queries with complexity $O(n\log(n))$ will lead to performance degradation. The query performance variations are also influenced by the change rate of the database table sizes.

This paper describes a framework to locate the performance-sensitive queries. A performance-sensitive query refers to a query which significantly impacts the service performance when the database size changes. The performance-sensitive queries are located based on the query complexity and the speed of the table size changes.

The performance-sensitive query locating process is then illustrated on three widely used web application benchmarks: RUBiS, RUBBoS, and TPC-W. A key observation is that performance-sensitive queries account for only a small fraction of the entire query set, less than 20%. Tests on TPC-W (the most challenging benchmark) to evaluate the effectiveness of this method on real life traffic demonstrate the effectiveness of this approach.

2 Related Work

The most successful database service products in the industry like Amazon RDS[12] and Microsoft SQL Azure[13] are provided. They offer cloud users database services in a pay-as-you-go mode. Consumers can seamlessly access databases through the Web without reference to the underlying IT infrastructures. However, RDS and Azure do not give performance promises for database applications hosted on their infrastructures. Existing commercial DaaS products and published research efforts have concentrated on availability and privacy concerns of database services. However, none have paid enough attention to performance or scalability assurance of database services.

Google has a system[14] that continuously monitors their database services and claims that the useful user activity statistical information for performance assurance only analyzes a small part of the entire monitored information set. The monitored data should be selected as early as possible to minimize the monitoring overhead by using load shedding to drop non-performance-sensitive queries, so that the overhead for transferring monitored data is kept small[15]. One common way to locate performance bottlenecks is to identify the most time-expensive queries. However, little work has been done to predict query performance for cloud database systems. Most systems simply passively detect outlier queries based on their execution time. For instance, if one query's execution time is 5 times larger than the average for all queries, then it is regarded as an outlier. The model in this paper can predict when a query will become an outlier in continuously changing database service systems using a database monitoring module that is seamlessly integrated into the database service environment.

As in database service systems, the searches in Information Retrieval (IR) systems can also be classified into easy queries (high average precision) and difficult queries (low average precision). Even when an IR system has good average execution time, some queries' performance can be very poor. Therefore, difficult queries must be identified early to handle them in advance[16]. Josiane and Ludovic[17] predicted query difficulty based on semantic features and related query difficulty. They claimed that the query difficulty was related to the query expression complexity. The method to locating performance-sensitive queries presented in this paper can also be applied to such circumstances.

3 Query Performance Predictions Based on Complexity

3.1 Query performance prediction based on basic operation complexity

Every query execution time can be logically expressed as a function of the database size involved in the

query execution plan, which is referred to as a complexity function. Moreover, a key observation is that all user query requests can be decomposed into atomic queries (e.g., relational operators in SQL or basic operations in cloud computing) which have the complexities $O(1)$, $O(\log(n))$, $O(n)$, $O(n\log(n))$, and $O(n^2)$[18, 19]. Atomic queries are SQL queries with one or no conditions after WHERE. For instance:

Query 1 SELECT count(*) FROM customer is $O(1)$ based on the count statistics or $O(\log(n))$ with a primary key or $O(n)$ without a primary key.

Query 2 SELECT i_stock FROM item WHERE i_id = N is $O(\log(n))$ with an index or $O(n)$ without an index on i_id.

Query 3 SELECT i_id FROM item is $O(n)$ without i_id index.

Query 4 SELECT * FROM item, author WHERE item.i_a_id = author.a_id is $O(n\log(n))$ or $O(n^2)$ based on the index information of the join attributes.

Query 1 with the count statistics has a time complexity of $O(1)$ since its execution time is constant and is unrelated to the actual table size because such queries do not need to scan the table or index, but only need to query the count statistics from the database dictionary information_schema. As such, the system can identify queries, execution time with complexity $O(1)$ by features like count, max, min, and no clauses after WHERE. The time complexity $O(\log(n))$ occurs when the involved query tuples can be scanned through the classical B-Tree index structure. In most cases, the B-Tree indexes have been built on the searched attributes either by the Database Administrator (DBA) manually or by DBMS automatically. For queries 1 and 2 with an index, the execution time complexity is $O(\log(n))$. If all the predicates after WHERE are in the form i_id = N, then this query's time complexity is $O(\log(n))$ or $O(n)$ based on its index information. Query 3 linearly scans the full table and retrieves every tuple in the table. Consequently, its time complexity is $O(n)$ and the query execution time increases linearly with table size. However, such queries are not common since they use linear table scans, which are extremely inefficient. Moreover, queries containing key words such as ORDER, GROUP or a predicate in the form of col1=col2, the minimum time complexity is $O(n\log(n))$ and the maximum time complexity is $O(n^2)$. Queries with time complexity of $O(n^2)$ seldom occur in real applications. In most cases, a B-Tree index will be built using the matched attributes so the complexity of the join operation becomes $O(n\log(n))$. Table 1 illustrates the time complexities of the most popular basic operations in SQL queries (n is the number of tuples in the underlying tables)[18, 19].

Thus, the query time complexity can be based on the query semantic structures so rules can be integrated into the DaaS program to analyze the query complexity automatically. For instance,

(1) If the query contains key words like ORDER, GROUP, and so on, then its complexity is $O(n\log(n))$.
(2) If the clauses after WHERE are all in the form of col1 = N or col1 = 'S', then the query's complexity is $O(\log(n))$ or $O(n)$.
(3) If the predicates after WHERE contain the equation in the form of col1= col2, then minimum complexity is $O(n\log(n))$ and the maximum complexity is $O(n^2)$.

These rules can be used to automatically analyze the query complexity from the semantic structure of user requests.

3.2 Predicting query execution time by combinations of the complexities

In real database services, few queries have only one basic complexity, and most queries are described by a combination of the complexities. Moreover, user query requests for database services include not only SQL queries, but also XQueries or DBMS associated stored procedures/functions. Consequently, the time complexity can only be inferred from the query semantics for only relatively simple structured SQL queries. Complex XQueries and stored procedures are assumed to include all five complexities with the complexity function weights found by leveraging monitored data points. The execution time of various types of queries can then be predicted.

The time complexity of $O(n\log(n))$ is a special case since it may change to $O(\log(n))$ due to the query's attribute distribution. The following is an example to

Table 1 Complexity of database operations.

Operation	Complexity
Select (with B-Tree index)	$O(\log(n))$
Select (without index)	$O(n)$
Join	$O(n\log(n))$
Order	$O(n\log(n))$
Group	$O(n\log(n))$
Cartesian product	$O(n^2)$

illustrate the complexity estimation process and the specific characteristics of time complexity $O(n\log(n))$:

a. SELECT * FROM order_line, item WHERE ol_o_id = N AND ol_i_id = i_id;

This query can be decomposed into two atomic queries:

b. SELECT * INTO order_line' FROM order_line WHERE ol_o_id = N;

c. SELECT * FROM order_line', item WHERE ol_i_id = i_id;

Query processing and optimization strategies first execute the selection operation which significantly reduces the table size in most cases. The output of the selection operation is called the *intermediate_relation*, which is used as the input to later operations. In this example, query b will be executed first and the output *intermediate_relation order_line*' will be used as the input to query c. The time complexity of query b is $O(\log(n))$ with an index or $O(n)$ without an index on ol_o_id while query c's time complexity is $O(n\log(n))$. Consequently, the time complexity for query a is $t = a \times \log(n) + b \times n \times \log(n) + c$ or $t = a \times n + b \times n \times \log(n) + c$. However, the time complexity of query c is influenced by the cardinality of *intermediate_relation*. The cardinality indicates the number of distinct values on attribute ol_o_id. If the cardinality of *intermediate_relation* is a constant (e.g., 1), which means that no matter how the size of table *order_line* changes, query b only retrieves one tuple for *intermediate_relation order_line*', and the time complexity of the following executed join operation changes from $O(n\log(n))$ to $O(\log(n))$. The reason is that one side of col1=col2 has a fixed size no matter how the database size changes. The most complex query can contain all five complexities.

After determining the query's complexity combination from its semantic structures, the system has to approximate the coefficients a, b, and c. The database can be populated with different scales with measure merits of their query execution time. Then, the coefficients can be approximated by a Nonlinear Least-Squares (NLLS) Marquardt-Levenberg algorithm with maintained data points[20]. The NLLS fits a set of data points with a curve function by minimizing the squared residuals between the function and the data points. In most cases, the system can decide the initial form of the curve (e.g., $y = a \times x^2 + b$) by analyzing the recorded data points. Then, NLLS solves for the coefficients to minimize $\sum_{i=1}^{N}[f(x_i) - y_i]^2$.

For example, the database can be populated with different scales (e.g., scale $1, 5, 10, 15, 20, 25,$ and 30) and the query execution time can be recorded (e.g., $t_1, t_5, t_{10}, t_{15}, t_{20}, t_{25}$, and t_{30}). The coefficients can be approximated by NLLS with the data point pairs (e.g., $(1, t_1)$, $(5, t_5)$, $(10, t_{10})$, $(15, t_{15})$, $(20, t_{20})$, $(25, t_{25})$, and $(30, t_{30})$).

4 Performance Sensitive Query Monitoring and Locating

The two key features of performance-sensitive queries are that the table sizes change quickly and the complexity function of queries on such tables include basic operations of complexities of $O(n)$, $O(n\log(n))$ or $O(n^2)$.

4.1 Changes monitoring of table size

In real applications, the sizes of some tables may increase much faster than others while some table sizes remain constant. Hence, a metric is defined to model the table size change rates. The fastest insert speed is used as the standard, with other table insert speeds expressed relative to this standard. For instance, if table item increases fastest with 100 tuple insertions in a time period, then it is used as the standard speed (e.g., *insertfactor* 1). If table customer has 20 inserted tuples in the same time period, this table's insert speed is *insertfactor* 0.2. The algorithm focuses on the tables which change the fastest, using a threshold (e.g., 0.1). If a table's *insertfactor* exceeds this threshold, then it is a critical table. Otherwise, the table's size is assumed to be constant since its insert speed is negligible compared with the other tables.

4.2 Performance-sensitive query locating

The queries having time complexity of $O(1)$ and $O(\log(n))$ never cause significant performance degradation due to related database table size changes. That time complexity of $O(1)$ indicates that the query execution time never changes with the table size. The time complexity of $O(\log(n))$ indicates that query execution time changes with the associated table size as $\log(n)$. One special property of the function $\log(n)$ is that when n is large enough, the value of $\log(n)$ remains almost constant. For example: $\log(8) = 3$ and $\log(2^{10}) = 10$. Furthermore, we observe that the queries with time complexity of $O(n^2)$ seldom occur in practice due to the rate of DBMS optimizations. Thus we need only focus on queries with time complexities of $O(n\log(n))$ and $O(n)$.

4.3 Case studies

The performance-sensitive query locating process was used to analyze three typical e-commerce benchmarks to evaluate its feasibility. The benchmarks were not run in real but the benchmarks were analyzed statically for the query semantic structures to get the information. The analysis shows that the performance-sensitive queries only need consider a small proportion of the entire query set, less than 20%.

4.3.1 RUBiS

RUBiS is an auction site benchmark modelled after eBay.com[21] which contains 9 tables, 26 distinct selection queries, and 10 Update/Delete/Insert (UDI) queries. Four table sizes, *categories*, *regions*, *old_items*, and *ids*, remain constant. The UDI queries usually do not contain ORDER, GROUP or join operations. Therefore, the complexities of the UDI queries are usually either $O(1)$ or $O(\log(n))$. Then, the algorithm needs only focus on the 26 selection queries. Six selection queries have time complexity $O(n\log(n))$ with relationship to tables *item*, *bids*, and *users*, whose sizes are not constant. Thus, all 6 selection queries are regarded as performance-sensitive queries. In RUBiS, performance-sensitive queries include only 16% (6/36) of the entire query set.

4.3.2 RUBBoS

RUBBoS is a bulletin-board benchmark modelled after slashdot.org[22] which contains 8 tables, 30 distinct selection queries, and 9 UDI queries. Table *categories* remains constant. Among the 30 selection queries, 12 queries have time complexity $O(n\log(n))$. Thus, the query semantic structures indicate 12 performance-sensitive queries. The 12 selection queries are related to the 6 tables, *stories*, *old_stories*, *submissions*, *comments*, *old_comments*, and *users*.

Other clues such as service description[21] can be leveraged to estimate the table size changes. The service description of RUBBoS indicates that the database administrators split both *stories* and *comments* into *new* and *old* tables for efficiency reasons. A deamon is periodically activated to move tuples of *stories* and *comments* from the new tables to the old tables so that the sizes of tables *comments* and *stories* will not become too large. Table *submissions* initially stores each submitted story, with the story later moved to table *stories*. Thus, table *submissions* is always relatively small. In addition, one user may submit dozens of *stories* and *comments*. Consequently, changes in table *users* are much slower than in table *old_comments* and *old_stories*. To summarize, the sizes of tables *stories*, *comments*, *submissions*, and *users* increase much slower than the tables *old_stories* and *old_comments*. Finally, the system need only focus on the $O(n\log(n))$ queries related to tables *old_stories* and *old_comments*. Thus, the RUBBoS benchmark has five performance-sensitive queries, only 13% (5/39) of the entire query set.

4.3.3 TPC-W

TPC-W is a widely used e-commerce benchmark that models an online bookstore like Amazon.com[23, 24]. The database contains 34 selection queries and 15 UDI queries. There are 10 tables in the database: *address*, *author*, *cc_xacts*, *country*, *customer*, *item*, *order_line*, *orders*, *shopping_cart*, and *shopping_cart_line*. The sizes of tables *item*, *author*, and *country* remain constant. 14 selection queries have time complexity $O(n\log(n))$ and 3 queries have time complexity $O(n)$. Thus, there are 17 candidates for performance-sensitive queries, but the constant sizes of tables *item* and *author* eliminates 7 candidates.

In addition, the name of each table in the database indicates its function which allows one to infer from the table names that some tables may change much faster than others. For example, table *customer* is used to store the customer information and table *shopping_cart* is used to store each online shopper's shopping cart. The sizes of tables *shopping_cart* and *shopping_cart_line* never increase significantly, since when each customer stops shopping, the shopping cart history is removed. Furthermore, the sizes of tables *customer* and *address* increase much slower than table *orders* for the same reason as with RUBBoS. Six performance-sensitive queries were located, only 12% (6/49) of the entire query set.

5 Experiments and Analysis

As noted, RUBiS and RUBBoS are relatively simple compared with TPC-W. Consequently, experiments were run using TPC-W, which is the most challenging of the three benchmarks. The first experiment was run for 144 hours to measure the table size changes. The results illustrate the performance-sensitive query locating process based on query complexity and table change rates. Finally, the complexity model is used to predict the performance-sensitive query performance in a mixed query

environment that is similar to real e-commerce applications.

5.1 Test setup

All the tests were performed on servers having Pentium(R) Dual-Core CPU 2.60 GHz, 2 GB memory, and 300 GB hard drives. Tomcat 6.0.24 was used as the application server, MySQL 5.1 as the database server, and Ubuntu 10.04 as the operating system. The query execution time was measured by MySQL admin tools. The servers were connected in a LAN; thus, the network latency between servers was negligible.

The program used the JAVA implementation of TPC-W from the University of Wisconsin[24]. The database was first populated with 10 000 tuples in table *item* and 144 000 tuples in table *customer*. Other table sizes were determined by these two tables according to the TPC-W specifications. The client workload was generated by the 500 remote browser emulators with a default think time of 7 s. The think time refers to the amount of time that the remote browser emulators wait between receiving a response and issuing another request. The whole process lasted for 144 hours. The time unit is "ms" unless otherwise specified.

TPC-W specifies three query mixes as *browsing*, *shopping*, and *ordering*. Unless otherwise specified, the experiments are based on "ordering" mix. The reason is that the tests are only to study query performance variations as the database size changes. The "ordering" mix is the most insertion-intensive mix, with the largest increases in the database sizes.

5.2 Table size changes in TPC-W

After running the test for 144 hours with the "ordering" mix, the table sizes increased:

table *address* increased 249 448 tuples,
table *cc_xacts* increased 3 878 520(*) tuples,
table *customer* increased 55 242 tuples,
table *order_line* increased 3 939 408(*) tuples,
table *orders increased* 3 878 535(*) tuples,
table *shopping_cart* increased 285 608 tuples,
table *shopping_cart_line* increased 62 510 tuples.

The results show that the change rates of the different tables vary significantly. The sizes of tables *order_line*, *orders*, and *cc_xacts* increase much faster (10 times or more), so these are the critical tables. The other table sizes were then treated as constants since their change rates are negligible compared to the critical tables. This results agree with the common sense. Section 4 showed that table *shopping_cart*, *shopping_cart_line*, *customer*, and *address* did not change much.

5.3 Performance-sensitive queries in TPC-W

Based on above discussion, the performance-sensitive query metrics of TPC-W are related to table *order_line*, *orders*, and *cc_xacts*, and then time complexity must be $O(n\log(n))$ or $O(n)$.

Note that this analysis only focuses on queries whose response times vary significantly when the database size changes. Other queries may perform poorly but their performance is not related to database size. The bottleneck queries refer to queries whose execution time is 3 times larger than the average or that may change significantly with the database size. For instance,

Query 5 SELECT item.i_id FROM item, author WHERE item.i_a_id = author.a_id AND substring(soundex(item.i_title),N,N) = substring (soundex('S'),N,N) ORDER BY item.i_title;

Query 6 SELECT i_id, i_title, a_fname, a_lname, SUM(ol_qty) AS val FROM tmpBestseller, order_line, item, author WHERE order_line.ol_o_id = tmpBestseller.o_id AND item.i_id = order_line.ol_i_id AND item.i_subject = 'S' AND item.i_a_id = author.a_id GROUP BY i_id ORDER BY val DESC LIMIT N,N;

Query 7 SELECT ol2.ol_i_id, SUM(ol2.ol_qty) AS sum_ol FROM order_line ol, order_line ol2, tmpAdmin WHERE ol.ol_o_id = tmpAdmin.o_id AND ol.ol_i_id = N AND ol2.ol_o_id = t.o_id AND ol2.ol_i_id <>N GROUP BY ol2.ol_i_id ORDER BY sum_ol DESC LIMIT N,N;

Query 8 SELECT o_id FROM customer, orders WHERE customer.c_id = orders.o_c_id AND c_uname = 'S' ORDER BY o_date, orders.o_id DESC LIMIT N,N.

Query 5 is a typical join operation example. The sizes of tables *item* and *author* remain constant due to TPC-W's specifications. Though this query's execution time is relatively long (e.g., 62 ms), the query 5 execution time does not change significantly as the database changes. Such query performance problems can be easily detected by the outlier threshold. For example, if the query execution time exceeds 50 ms, the query will be identified as a performance outlier. Also, such queries do not complicate performance predictions. Therefore, such queries are ignored, because the main topic of this paper is not to detect performance outliers but to proactively predict query performance bottlenecks as the database changes.

In addition, the queries related to temporary tables whose sizes are always constant are excluded along with the queries whose time complexities change from $O(n\log(n))$ to $O(\log(n))$ due to the special cardinality of the intermediate_relation. For instance:

SELECT orders.*, customer.*, cc_xacts.cx_type, ship.addr_street1 AS ship_addr_street1, ship.addr_street2 AS ship_addr_street2, ship.addr_state AS ship_addr_state, ship.addr_zip AS ship_addr_zip, ship_co.co_name AS ship_co_name, bill.addr_street1 AS bill_addr_street1, bill.addr_street2 AS bill_addr_street2, bill.addr_state AS bill_addr_state, bill.addr_zip AS bill_addr_zip, bill_co.co_name AS bill_co_name FROM customer, orders, cc_xacts, address AS ship, country AS ship_co,address AS bill, country AS bill_co WHERE **orders.o_id = ?** AND cx_o_id = orders.o_id AND customer.c_id = orders.o_c_id AND orders.o_bill_addr_id = bill.addr_id AND bill.addr_co_id = bill_co.co_id AND orders.o_ship_addr_id = ship.addr_id AND ship.addr_co_id = ship_co.co_id AND orders.o_c_id = customer.c_id.

The query semantic structures suggest that the complexity of this query should be $O(n\log(n))$. However, if the cardinality of orders.o_id is n, which indicates orders.o_id = ? only retrieves one tuple for the following join operations, then the time complexity of this query changes from $O(n\log(n))$ to $O(\log(n))$. This example also illustrates that complex query semantic structures do not always lead to complex queries complexity. The result was three performance-sensitive queries in TPC-W, queries 6-8.

5.4 Effectiveness analysis of complexity model

This subsection uses the complexity model to predict the performance of three performance-sensitive queries in TPC-W. The size of the initially populated database is ignored because it is much smaller than the database size changes (1/30).

As shown in Fig. 1, the query 8 execution time increases with the table *orders* size. However, this figure does not suggest any complexity pattern. Consequently, the data points are analyzed statistically. All three tables have approximately 3 900 000 inserted tuples. Without loss of generality, 120 000 tuples are defined as *basicScale* (scale 1). Then, average query execution time is calculated for scale 5, 10, 15, 20, 25, and 30. The three query times were calculated as $t = a \times \log(n) + b \times n\log(n) + c$, with the coefficients determined by the NLLS algorithm. Finally, the parameterized complexity

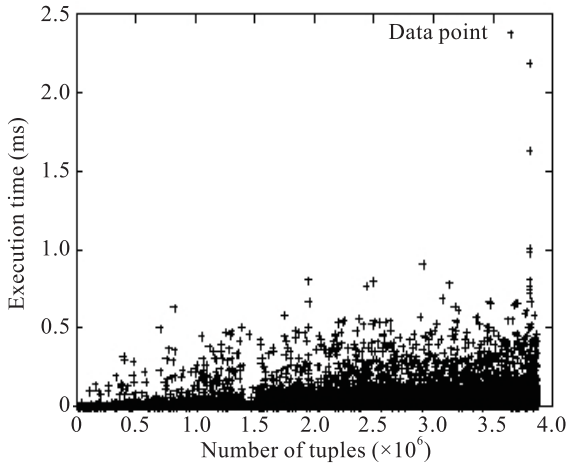


Fig. 1 Execution time for query 8 at different table sizes.

function was used to predict the query execution time for an arbitrary table size (e.g., scale 30).

$$\text{ErrorRate} = \frac{|\text{PredictedValue} - \text{TrueValue}|}{\text{TrueValue}} \quad (1)$$

The following details the data points processing strategy with scale 10 as an example. The algorithm maintains the number of inserted tuples. Whenever this number reaches $10 \times basicScale$, the execution time of the following 1000 queries are used to calculate the average. The execution time for the 1000 queries is sorted with only 900 data points in the middle of the distribution used. The averages at scales 1, 10, 15, 20, and 25 are used by the NLLS algorithm to fit the complexity function. The result is used to predict the query execution time at scale 30. The error rate was calculated by Eq. (1), with the error rates of queries 6-8 being 3.9%, 4.2%, and 9.0%. The calculated data points and the fitted complexity function at different scales are shown in Fig. 2.

The data was also analyzed be defining 120 000 tuples as *basicScale*, with the query execution time calculated every $n \times basicScale$ ($1 \leqslant n \leqslant 30$). When the number of inserted tuples reached $n \times basicScale$, the system analyzed 500 queries before this point and 500 queries after the point. The reason is that when query 7 executed 1000 times, approximately 90 000 tuples were inserted into table *order_line*. The first 29 data points were used to fit the complexity function, with this function then used to predict the query execution time at scale 30. As shown in Fig. 3, this curve also accurately fits the points. The prediction error rates for the three queries were 3.48%, 2.74%, and 12.5%.

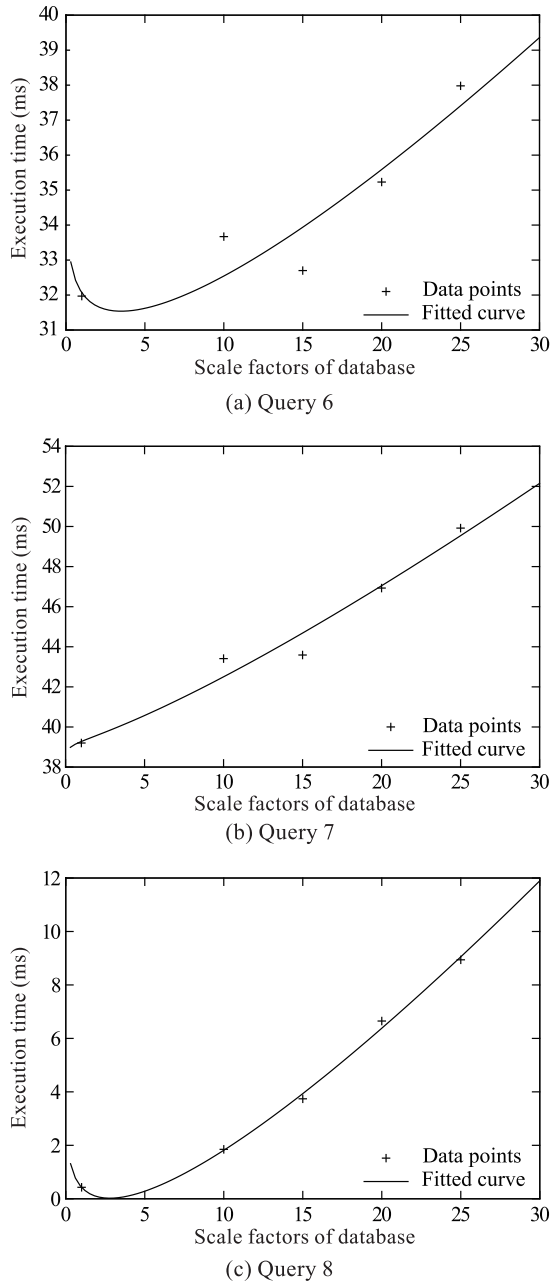


Fig. 2 Execution time for queries 6-8 at different scales (5 points).

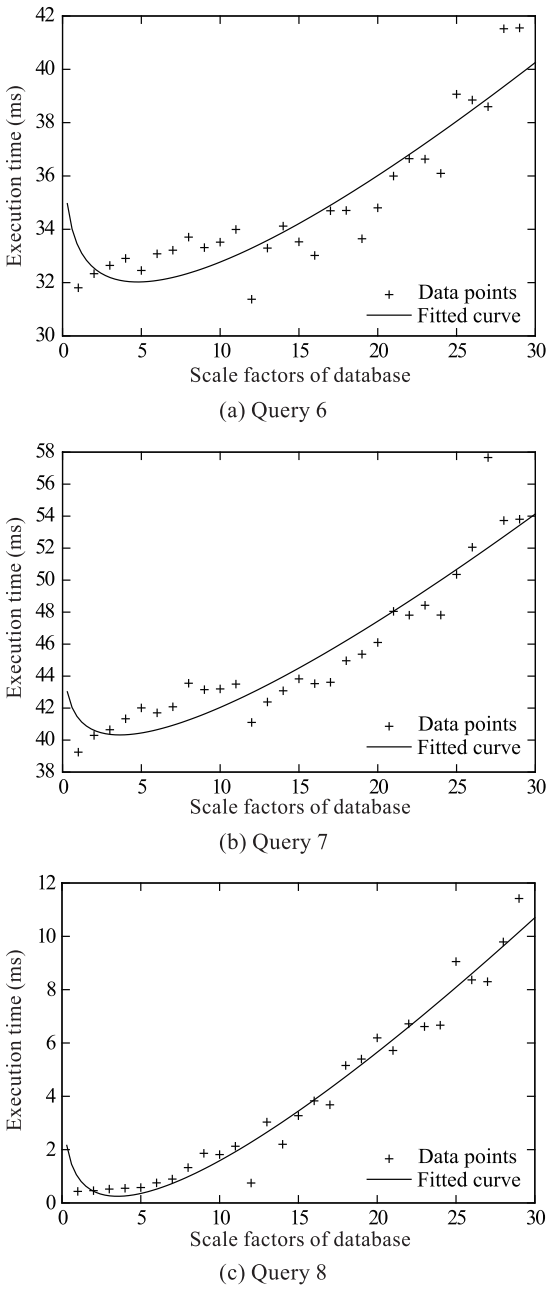


Fig. 3 Execution time for queries 6-8 at different scales (29 points).

5.5 Predicting query performance using a linear function

One limitation of the NLLS algorithm is that the relative scale of database must be known to fit the complexity function. In real cloud applications, the service consumers simply subscribe the database infrastructure from the service provider, with the data itself treasured as valuable, sensitive assets; thus, the database owner is reluctant to share it with third parties. Consequently, the initial database may appear as a black-box to the SP. If the initial database size is unpublished and can not be ignored, SP cannot calculate the relative scale from only the database monitoring log. For instance, the log will show that this table has received 100 000 tuples, but the system cannot decide whether this table has increased 5 fold or 2 fold. Thus, the SP must predict query performance with only the knowledge in the database logs in the cloud

environment.

This study has shown that performance-sensitive queries all have time complexities of $O(n\log(n))$ or $O(n)$. Therefore, the analysis uses the special property of the function $\log(n)$, that the function tends to become constant as n increases. Thus, the complexity function $y = a\log(n) + bn\log(n) + c$ can be simplified to $y = an + b$ which is also the complexity function of $O(n)$. This result is in line with result in Section 5.4. Consequently, the data points at scales 10, 15, 20, and 25 can be used to fit the complexity function $y = an + b$ to predict the query execution time at scale 30. The prediction error rates of queries 6-8 were 2.3%, 2.4%, and 4.2% with this method.

5.6 Discussion

This work didn't include comparisons with other query performance prediction models, which predict query performance from SQL text structures[25] or I/O-based metrics[26]. Marquardt[27] presented statistical models to predict system resource utilization for highly-concurrent OLTP workloads. They developed black-box models to estimate the utilization of linear resources (e.g., CPU) and non-linear resources (e.g., disk I/O) based on past resource utilization statistics and linear regression techniques. Moreover, they tried to classify the transactions into various types based on their semantics and studies the resource utilization changes for different ratios of transaction types. Although the comment paper mainly focuses on resource utilization estimates, much was learned from the authors about statistical regression and the classification of transaction types. Mozafari et al.[28] presented a prototype system called "DBSeer which addresses the problem of resource and performance predictions for a given OLTP database workload". DBSeer utilizes simple linear techniques to model the CPU resources and Monte-Carlo simulations to model the I/O resources. DBSeer then clusters database transactions into classes of similar templates.

However, most existing models assume that the database size is constant forever with changes in workloads or environmental parameters (e.g., upgrades, schema changes)[29]. The current work differs from such models with analysis of what happens if above assumptions are not satisfied. Thus, the assumption and definition of the problem have changed. Their problem is to study the system performance for different workloads (mostly read only) while the current problem is to model performance variations for changing database sizes. To the best of our knowledge, no other published efforts have focused on this problem.

Further work will use a large cloud environment (Amazon EC2) for experiments. One of biggest issues for deploying an application in the cloud is the performance variations of VM. Future work will also take such variations into account. EC2 provides different classes of VMs whose capacities vary greatly, so this model can be deployed on different classes of VMs to clarify whether the result still holds for different VM configurations.

6 Conclusions

Most query performance predictions have assumed that the database size is constant and predict the performance for various query features. This paper demonstrates that the relationship between query execution time and database size can be captured by query's complexity functions. While such query performance predictions provide performance gains by allowing service providers to detect performance bottlenecks in advance.

Performance-sensitive query locating exploits the fact that database's performance bottlenecks are usually just a small fraction of the entire query set. This strategy is used to analyze three typical benchmarks while the analysis of TPC-W focused on the performance of only three performance-sensitive queries. Although this approach was verified for all three applications, one cannot exclude the existence of applications whose queries are all performance-sensitive. This may be the case for OLAP applications. In such cases, this method should be invested more in cloud database applications in future.

Acknowledgements

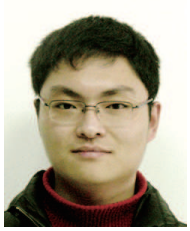
We thank Shuo Chen and Dejun Jiang for their generous support.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, Above the clouds: A berkeley view of cloud computing, *Communications of The ACM - CACM*, vol. 53, no. 4, pp. 50-58, April 2010.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the

5th utility, *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599-616, June 2009.
[3] Amazon EC2 website, http://aws.amazon.com/ec2/, 2009.
[4] Forrester Research Website, http://www.akamai.com/2seconds, 2009.
[5] Z. Wei, J. D. Jun, C. C. Hung, and M. V. Steen, Service-oriented data denormalization for scalable web applications, in *Proc. 17th Int. World Wide Web Conf.*, Beijing, China, 2008, pp. 267-276.
[6] C. Loboz, S. Smyl, and S. Nath, DataGarage: Warehousing massive performance data on commodity servers, in *Proc. 36th VLDB Endowment*, Singapore, 2010, pp. 1447-1458.
[7] J. D. Jun, Performance guarantees for web applications, Ph.D. Dissertation, Dept. Computer Science, Vrije Universiteit, Amsterdam, 2011.
[8] E. M. Voorhees, The TREC robust retrieval track, in *Proc. 28th Annu. Int. ACM SIGIR Conf.*, Salvador, Brazil, 2005, pp. 11-20.
[9] D. Florescu and D. Kossman, Rethinking cost and performance of database systems, in *Proc. 35th ACM SIGMOD Int. Conf. on Management of Data*, Indiana, USA, 2009, pp. 43-48.
[10] Q. Guo, R. W. White, S. T. Dumais, J. Wang, and B. Anderson, Predicting query performance using query, result, and user interaction features, in *Proc. 9th RIAO Conf. Adaptivity, Personalization and Fusion of Heterogeneous Information*, Paris, France, 2010, pp.198-201.
[11] C. D. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, Gigascope: A stream database for network applications, in *Proc. 29th ACM SIGMOD Int. Conf. on Management of Data*, Rhode Island, USA, 2003, pp. 647-651.
[12] Amazon RDS, http://aws.amazon.com/rds/, 2009.
[13] Microsoft SQL Azure, http://www.microsoft.com/windowsazure/features/database/, 2011.
[14] S. Chaudhuri, A. C. Konig, and V. Narasayya, Database monitoring system, US Patent: 7194451B2, Feb.26, 2004.
[15] C. Donald, C. Ugur, C. Mitch, C. Christian, L. Sangdon, S. Greg, S. Michael, and T. Nesime, Monitoring streams C a new class of data management applications, in *Proc. 28th Int. Conf. on Very Large Data Bases*, Hong Kong, China, 2002, pp. 215-226.
[16] E. Y. Tov, S. Fine, D. Carmel, and A. Darlow, Learning to estimate query difficulty, in *Proc. 28th Annu. Int. ACM SIGIR Conf.*, Salvador, Brazil, 2005, pp. 512-519.
[17] M. Josiane and T. Ludovic, Linguistic features to predict query difficulty —- A case study on previous TREC campaigns, presented at ACM SIGIR'05 Query Prediction Workshop on Predicting Query Difficulty —- Methods and Applications, Bahia, Brazil, 2005.
[18] H. G. Molina, J. D. Ullman, and J. Widom, *Database System Implementation*. Prentice Hall, 1999.
[19] O. M. Tamer and V. Patrick, *Principles of Distributed Database Systems*. Prentice Hall, 1999.
[20] B. Mozafari, C. Curino, A. Jindal, and S. Madden, Performance and resource modeling in highly-concurrent OLTP workloads, in *Proc. 37th ACM SIGMOD Int. Conf. on Management of Data*, New York, USA, 2013, pp. 330-341.
[21] RUBiS, http://rubis.ow2.org/, 2004.
[22] RUBBoS, http://jmob.ow2.org/rubbos.html, 2005.
[23] TPC-W Specification, Version 1.8, http://www.tpc.org/tpcw/, 2000.
[24] TPC-W Implementation, http://www.ece.wisc.edu/∼pharm/tpcw.shtml, 2002.
[25] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson, Predicting multiple metrics for queries: Better decisions enabled by machine learning, in *Proc. 25th Int. Conf. on Data Engineering*, Shanghai, China, 2009, pp. 592-603.
[26] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, Performance prediction for concurrent database workloads, in *Proc. 37th ACM SIGMOD Int. Conf. on Management of Data*, Athens, Greece, 2011, pp. 337-348.
[27] D. W. Marquardt, An algorithm for least-squares estimation of nonlinear parameters, *Siam Journal on Applied Mathematics*, vol. 11, no. 1, pp. 431-441, 1963.
[28] B. Mozafari, C. Curino, and S. Madden, DBSeer: Resource and performance prediction for building a next generation database cloud, in *Proc. 6th Biennial Conf. on Innovative Data Systems Research*, California, USA, 2013, pp. 162-165.
[29] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu, Oracle's SQL performance analyzer, *IEEE Data(base) Engineering Bulletin*, vol. 31, no. 1, pp. 51-58, March, 2008.



Ye Zhou is a student in the Department of Computer Science and Technology at Tsinghua University (China). He focuses on the area of cloud computing, database as a service, service performance, and system architecture. He received his BEng degree from Southeast University (China) in 2009.

Xiaojun Ye received his BS degree in mechanical engineering from Northwest Polytechnical University, Xi'an, China, in 1987 and PhD degree in information engineering from INSA Lyon, France, in 1994. Currently, he is a professor at School of Software, Tsinghua University, Beijing, China. His research interests include cloud data management, data security and privacy, and database system testing.

Chihung Chi is currently a professor in the School of Software, Tsinghua University. He obtained his PhD degree from Purdue University in 1991. After working in Philips Laboratories and in IBM Poughkeepsie, he returned to academia (firstly to Chinese University of Hong Kong, then National University of Singapore and now, with Tsinghua University). He has published about 200 papers in international conferences and journals and holds 6 U.S patents. He is the program/general chairman of WCW 2004, AWCC 2004, IEEE SOSE 2006, ICSOC 2009, SCC 2009, SIE 2010, EDOC 2011, EDOC 2012, and CWI 2012. His main research areas include software engineering, service engineering, cloud computing, intelligence and analytics, content networking, distributed computing, and system security.