Bulletin of the Technical Committee on

Data Engineering

March 2008 Vol. 31 No. 1

IEEE Computer Society

Letters

The TC on Data Engineering
Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is
`http://tab.computer.org/tcde/index.html`.

The Data Engineering Bulletin
The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at
`http://tab.computer.org/tcde/bull_about.html`.

Letter from the Editor-in-Chief

Bulletin Announcement

In the last issue of the Bulletin, I very proudly announced that all issue of the Bulletin are now available on the Bulletin web site. I now want to announce changes to the web site itself. For this effort, I want to thank Thomas Risse, who is the Secretary/Treasurer of the TC on Data Engineering. Thomas does much more than this, including much of the administrative work for the ICDE Steering Committee. With respect to the web site, Thomas has reorganized and designed an integrated web site, including information about the TC on Data Engineering, the ICDE Conference, and the Data Engineering Bulletin. This effort brings together the database activity of the Computer Society into one integrated web site.

The new web site is hosted by the IEEE Computer Society. All the web pages mentioned earlier are linked together. The URL for the TC on Data Engineering is `http://tab.computer.org/tcde/index.html`; for the Data Engineering Bulletin, it is `http://tab.computer.org/tcde/bull_about.html`. The new Bulletin main web page links to newly designed web pages listing the issues, though these web pages do not change their URLs, nor does the location for the issues themselves. The earlier main Bulletin web pages will be phased out, but currently redirect browsers to the new page. Additional changes will surely come over time, so stay tuned.

I encourage all of you to explore the new web site. Your feedback is surely welcome. And thanks again to Thomas Risse for making this happen.

The Current Issue

Many in the database field were surprised 20 years ago when the TPC benchmarks first emerged as standards for comparing database performance. The differences in performance of commercial database systems on TPC-A and TPC- B, the debit/credit benchmarks, were substantial, both in terms of cost and peak performance. The TPC benchmarks, both these early ones and the several later ones, have had a wonderful impact in improving database products and is yet another example of our debt to Jim Gray, who played a very large role in getting the benchmarking efforts started. The result of this is that all commercial database systems have improved enormously. And this is not simply that these systems rode the wave of hardware improvements. Much more has happened. Now, all commercial database vendors have sizable benchmarking and testing groups. And these groups have done much to improve both the quality and the performance of the database products.

The current issue explores not the performance of database systems so much as the strategies and techniques used in efforts to enhance the ability to test correctness and improve performance. This is an issue in which the majority of the articles come from people working for industrial vendors of database products and applications, but leavened with research articles as well. Publication of this combination of industrial practice and research on an important topic for our field is a real strength of the Data Engineering Bulletin and an important part of what I consider to be its charter.

Jayant Haritsa has brought together this interesting collection of articles from major database, application, and hardware vendors, with university researchers that explores how databases are tested and tuned. I think that as you read this issue, you will be impressed by the technology, effort, and insights that have gone into these efforts. The success of efforts like those described here is demonstrated regularly via the continuous improvement that we see in the products offered by the database industry. I want to thank Jayant for his very successful effort in producing this issue.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Today's industrial-strength database engines, both commercial and public-domain, are designed to provide highly sophisticated functionalities, making them the backbone of the information society. Not surprisingly, a fallout of this sophistication is that the internal software infrastructure has become extremely complex, making it a technically challenging task to (a) verify the correctness of the engine components, and (b) tune the system to meet the desired performance objectives.

This issue of the Data Engineering Bulletin describes both novel research proposals and current industrial practices for the testing and tuning of enterprise database systems. Historically, these topics have received comparatively little attention in the research literature. However, there has been growing awareness in recent years of the rich set of problems on offer, which are simultaneously technically challenging and of immediate practical relevance, as exemplified by the articles featured in this issue.

The first article, by Debnath, Mokbel and Lilja from U. of Minnesota, considers the problem of efficently selecting, from among a dauntingly large number, the most relevant configuration parameters for tuning a database engine. They employ an experimental design methodology that makes the computational effort linear in the number of parameters, and quantitatively demonstrate with an implementation on PostgreSQL, that this approach is capable of successfully identifying the critical parameters.

The second article by Mehta, Gupta, Wang and Dayal of HP Labs presents a holistic machine-learning-based approach towards workload management for enterprise data warehouses addressing issues of admission control, scheduling and progress monitoring. Their techniques have been evaluated on real-world warehousing environments with promising results.

The third article by Krompass, Scholz, Albutiu and Kemper from TU Munich, and Kuno, Wiener and Dayal from HP Labs, investigates the specification and satisfaction of quality-of-service objectives in the context of operational data stores hosting workloads with a mix of transactional and decision support queries. They also survey the infrastructure provided by current industrial products to support these objectives.

The fourth article by Binnig, Kossmann and Lo from SAP, ETH and Hong Kong Polytechnic U., respectively, brings a fresh outlook to the design of test databases through the use of database techniques such as declarative specifications and logical data independence. Specifically, they advocate the ideas of "reverse query processing" (given a schema, a query and a result, generate a compliant database), and "symbolic query processing" (the database consists of symbolic, rather than concrete, data), which can be used to test engine components.

We then have a series of articles on current industry practices. First, Giakoumakis and Galindo-Legaria from Microsoft provide a guided tour through the arduous world of testing database query optimizers. They also overview the array of techniques used in testing the SQL Server optimizer. Then, Joshi, Lamb and Sandstrom from Oracle present the testing tools utilized for the Berkeley DB family of database engines, their task rendered additionally difficult because users, taking advantage of the source-code availability, may either modify or port the engines to new platforms. Next, Yagoub, Belknap, Dageville, Dias, Joshi and Yu from Oracle present the SQL performance analyzer implemented in Oracle 11g to help users investigate "what-if" scenarios by forecasting and analyzing the impact of system changes on SQL workload performance before deployment. Finally, Gittens, Gupta, Godwin, Pereyra and Riihimaki of IBM, tackle the notoriously tricky problem of catching timing-related errors and defects in complex multi-threaded systems such as database engines. Their proposed technique attempts to trigger unexpected behavior by iteratively executing system tests with a background workload.

In closing, we thank all the article authors for their painstaking and timely efforts in developing their contributions for this special issue. Our hope is that the work presented here will serve as a strong stimulus for the academic and industrial research communities to address, with renewed vigor and resources, the development of stable and efficient database engines.

Jayant R. Haritsa
Indian Institute of Science, Bangalore

Exploiting the Impact of Database System Configuration Parameters: A Design of Experiments Approach

Biplob K. Debnath, Mohamed F. Mokbel, and David J. Lilja
University of Minnesota, Twin Cities, USA.
debna004@umn.edu, mokbel@cs.umn.edu, and lilja@ece.umn.edu

Abstract

Tuning database system configuration parameters to proper values according to the expected query workload plays a very important role in determining DBMS performance. However, the number of configuration parameters in a DBMS is very large. Furthermore, typical query workloads have a large number of constituent queries, which makes tuning very time and effort intensive. To reduce tuning time and effort, database administrators rely on their experience and some rules of thumb to select a set of important configuration parameters for tuning. Nonetheless, as a statistically rigorous methodology is not used, time and effort may be wasted by tuning those parameters which may have no or marginal effects on the DBMS performance for the given query workload. Database administrators also use compressed query workloads to reduce tuning time. If not carefully selected, the compressed query workload may fail to include a query which may reveal important performance bottleneck parameters. In this article, we provide a systematic approach to help the database administrators in tuning activities. We achieve our goals through two phases. First, we estimate the effects of the configuration parameters for each workload query. The effects are estimated through a design of experiments-based PLACKETT & BURMAN *design methodology where the number of experiments required is linearly proportional to the number of input parameters. Second, we exploit the estimated effects to: 1) rank DBMS configuration parameters for a given query workload based on their impact on the DBMS performance, and 2) select a compressed query workload that preserves the fidelity of the original workload. Experimental results using PostgreSQL and TPC-H query workload show that our methodologies are working correctly.*

1 Introduction

Businesses are increasingly building larger databases to cope with the rapid current growth of data. Consistent performance of the underlying database system is key to success of a business. A typical database management system (DBMS) has hundreds of configuration parameters and the appropriate setting of these parameters plays a critical role in performance. Database administrators (DBAs) are expected to tune the configuration parameters to appropriate values that get the best DBMS performance for the application of interest. The success of tuning depends on many factors including the query workload, relational schemas, as well as the expertise of the DBAs [20]. However, skilled DBAs are becoming increasingly rare and expensive [16]. A recent

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

study on information technology versus DBA costs showed that personnel cost is estimated at 47% of the total cost of ownership [13]. As has been recenlty reported, DBAs spend nearly a quarter of their time on tuning activities [20]. To reduce the total cost of ownership, it is of essence that DBAs focus only on tuning those configuration parameters which have the most impact on system performance for a representative query workload.

Different database configuration parameters have different impact on a DBMS performance. A sound statistical methodology for quantifying the impact of each configuration parameter and the interactions among these parameters on a DBMS performance is to perform a *full factorial design* [17], where all possible combinations of the input values of the configuration parameters are considered. However, the major problem in applying a *full factorial design* in a DBMS is the large number of configuration parameters. For example, PostgreSQL [1] has approximately 100 configuration parameters and all parameters have multiple possible values. Even if each configuration parameter assumes only two values, then, given a query workload of q queries, we have to perform $q * 2^{100}$ experiments at least twice to apply a full factorial design, which is not feasible in terms of time and effort. To avoid this problem, in many cases, DBAs rely on their experience and rules of thumb to select the appropriate values for the configuration parameters. As heuristics based on experience and intuition are often used, time and effort may be wasted to enhance the performance by tuning those parameters that may have no or marginal effects on the overall performance of the given query workload. In general, misdirected tuning efforts increase the total cost of ownership [6, 8, 12, 14].

In this article, we are addressing the following problem: *Given a DBMS, a set of configuration parameters, a range of values for all parameters, and a query workload; estimate the effect of each configuration parameter based on its impact on the DBMS performance for the query workload*. In particular, we propose a methodology based on the PLACKETT & BURMAN (P&B) design [19] to estimate the impact of database system configuration parameters. The main idea is to conduct a linear number of experiments that provide an approximate sampling of the entire search space. In each experiment, the values of the configuration parameters are varied systematically over a specified range of acceptable values. Subsequent analysis of the collected experimental data is used to estimate the effects of the configuration parameters on the DBMS performance for the given query workload. Once we have the estimated effect of each configuration parameter for all workload queries, we can exploit these effects for: (1) ranking the configuration parameters based on the impact on DBMS performance for the entire query workload, and (2) selecting a compressed query workload based on the similarities of performance bottleneck parameters that preserves the fidelity of the original workload.

The rest of this article is organized as follows: Section 2 describes our *design of experiments*-based methodology. The methodology to estimate the effects of the configuration parameters is described in Section 3. Ranking configuration parameter and selecting a compressed workload are explained in Section 4. Experimental results are described in Section 5. Section 6 highlights related work. Finally, Section 7 concludes the article.

2 Design of Experiments Based Methodology

The simplest design strategy to quantify the impact of all factors and interactions is to apply a *full factorial design*, for example, ANOVA [17], in which system response is measured for all possible input combinations. However, a *full factorial design* requires an exponential number of experiments. To reduce the number of experiments, we make two assumptions. First, the provoked response, such as the total execution time, is a monotonic function of the input parameter values. This indicates that for each configuration parameter, we can consider only two values: minimum and maximum. The intuition behind this is that stimulating the system with inputs at their extreme values will provoke the maximum range of output responses for each input. Second, according to the *sparsity of effects principle*, system response is largely dominated by a few main factors and low-order interactions. As a consequence, we can safely ignore the effects of higher order interactions. Based on these assumptions, we use a *two-level factorial* design methodology named PLACKETT & BURMAN (P&B) design [19], which requires only linear number of experiments.

	P&B Design Matrix							**Execution Time**		
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	**Q1**	**Q2**	**Q3**
Exp_1	+1	+1	+1	-1	+1	-1	-1	34	110	10.2
Exp_2	-1	+1	+1	+1	-1	+1	-1	19	72	10.1
Exp_3	-1	-1	+1	+1	+1	-1	+1	111	89	10.3
Exp_4	+1	-1	-1	+1	+1	+1	-1	37	41	10.3
Exp_5	-1	+1	-1	-1	+1	+1	+1	61	96	10.2
Exp_6	+1	-1	+1	-1	-1	+1	+1	29	57	10.2
Exp_7	+1	+1	-1	+1	-1	-1	+1	79	131	10.3
Exp_8	-1	-1	-1	-1	-1	-1	-1	19	47	10.1
Exp_9	-1	-1	-1	+1	-1	+1	+1	135	107	10.3
Exp_{10}	+1	-1	-1	-1	+1	-1	+1	56	74	10.3
Exp_{11}	+1	+1	-1	-1	-1	+1	-1	112	48	10.1
Exp_{12}	-1	+1	+1	-1	-1	-1	+1	74	91	10.1
Exp_{13}	+1	-1	+1	+1	-1	-1	-1	55	99	10.3
Exp_{14}	-1	+1	-1	+1	+1	-1	-1	117	123	10.1
Exp_{15}	-1	-1	+1	-1	+1	+1	-1	51	77	10.3
Exp_{16}	+1	+1	+1	+1	+1	+1	+1	76	81	10.2

Table 1: The P&B design matrix with foldover for N = 7. Execution time of queries Q1-Q3 are in the last three columns.

For each experiment of the P&B design, the value of each parameter is given by a prescribed *P&B design matrix*. Table 1 gives an example of the *design matrix* for the seven parameters P_1, P_2, P_3, P_4, P_5, P_6, and P_7 depicted by the columns 2-8. The Exp_i indicates the values of the configuration parameters that will be used in the i-th experiment. An entry in the parameter columns of the design matrix is either "+1" or "-1", that corresponds to a value slightly higher or lower than the normal range of values for the corresponding parameter, respectively. The "+1" and "-1" values are not restricted to only numeric values. For example, for the buffer page replacement algorithm, the "-1" value can be "RANDOM" and "+1" value can be "CLOCK". The *P&B design matrix* is constructed by cyclic repetition of a single series using a simple methodology. It has been verified that such a method would result in desirable statistical properties [4]. Also, it has been verified that if the monotonic and low interactions assumptions are valid, the P&B design generates comparable results as the *full factorial design*. The detailed theoretical explanation behind this behavior is explained in [19].

The dimensions of the *P&B design matrix* depend on the number of configuration parameters, N. The base design matrix has X rows and $X-1$ columns, where X is the next multiple of four greater than N, i.e., $X = \lfloor (N/4)+1 \rfloor * 4$. For example, if N = 7, then X = 8, while if N = 8, then X = 12. If $N < (X-1)$, then the number of columns in the *P&B design matrix* is more than the number of configuration parameters. In this case, the additional $(X-N-1)$ last columns of the *P&B design matrix* are simply ignored. The recommendations of the "+1" and "-1" parameter value settings for X = 8, 12, 16, . . ., 96, 100 experiments are given in [19]. The first row of the *P&B design matrix* is selected based on those recommendations according to the value of X. The rest $(X-1)$ rows of the *P&B design matrix* are constructed by right cyclic shifting of the immediate preceding row. All entries of the Exp_X-th row of the P&B design matrix are set to "-1". The columns 2-8 in the first eight experiments (Exp_1-Exp_8) of the Table 1 indicates the base *P&B design matrix* for N=7.

An improvement of the base *P&B design* methodology is the *P&B design with foldover* [18]. The *foldover* helps to quantify the two parameter interactions more accurately. However, it requires X additional experiments. The additional rows in the *P&B design matrix* are constructed by reversing the sign of the top X rows matrix entries. The last eight rows (Exp_9-Exp_{16}) of Table 1 gives the additional design matrix entries for the *foldover* for N= 7. Experiments are conducted by setting up the values of the configuration parameters according to the *P&B design matrix* and response time is recorded to estimate the effect of each parameter.

	P&B Effect							COV of Execution Times
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	
Q1	109	79	167	193	21	25	177	0.55
Q2	61	161	9	143	39	185	109	0.32
Q3	0.40	0.80	0.00	0.40	0.40	0.00	0.40	0.01

Table 2: The P&B effects for the queries Q1, Q2, and Q3.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Q1	0.56	0.41	0.87	1.0	0.11	0.13	0.92
Q2	0.33	0.87	0.05	0.77	0.21	1.0	0.59

Table 3: The P&B normalized effects with respect to the maximum effect for the queries Q1 and Q2.

3 Effect Estimation of the Configuration Parameters

This section describes how to use the *P&B design* methodology described in Section 2 to estimate the effects of configuration parameters for each query of the given workload. The effect of each configuration parameter is calculated by multiplying the corresponding “+1” or “-1” of that parameter in the Exp_i-th row of the *P&B design matrix* with the query execution time of the i-th experiment, and summing up the products across all rows of the design matrix. The absolute value of the net effect is used in the subsequent analysis.

For illustration, suppose we estimate the P&B effects of a query workload consisting of three queries Q1, Q2, and Q3 as listed in Table 1. We have seven configuration parameters, P_1 to P_7. In this example, we assume that *foldover* is used, therefore we conduct 16 experiments. The specification of the parameter values that need to be used in all 16 experiments are given in columns 2-8 and rows Exp_1-Exp_{16} of Table 1. The net effect of the first parameter P_1 for query Q1 is calculated by multiplying the entries in the second column with the entries in the ninth column and summing up across all 16 rows (Exp_1-Exp_{16}). For query Q1, the net effect of the parameter P_1 is estimated as: $Effect_{P_1} = abs((+1*34)+(-1*19)+\ldots+(-1*51)+(+1*76)) = abs(-109) = 109$. Similarly, the net effect of the second parameter P_2 for query Q1 is calculated by multiplying the entries in the third column with the entries in the ninth column and summing across all 16 rows (Exp_1-Exp_{16}), and so on. Table 2 gives the net P&B effects of all seven parameters for the queries Q1, Q2, and Q3.

The last column of Table 2 gives the *coefficient of variation (COV)* of the response time across all experiments for queries Q1, Q2, and Q3. COV is defined as the ratio of the *standard deviation* to the *average* execution time. A very low COV value indicates that all effects are essentially the same, i.e., the query performance will not be affected by the change in configuration parameters settings. In general, if the COV is less than 0.05, we can safely ignore the effects and mark the corresponding query as tuning insensitive. In the illustrative example, query Q3 is tuning insensitive as its COV is 0.01.

4 Exploiting the Configuration Parameters Effects

Once we have the P&B effects of the configuration parameters for all workload queries, we can use these estimated effects to: (1) rank the configuration parameters for the entire workload based on their relative impact on the DBMS performance, and (2) select a compressed query workload that preserves the fidelity of the original query workload. We describe these two methodologies in detail in the following subsections. Throughout this section, we will use the query workload in Table 1 as a running example.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Q1	3	4	1	1	7	7	1
Q2	5	1	7	2	6	1	3

Table 4: Ranking of the configuration parameters for the queries Q1 and Q2.

4.1 Ranking the Configuration Parameters for a Query Workload

Ranking configuration parameters for a query workload consists of two steps. First, we rank the parameters for each tuning sensitive query of the workload based on the relative magnitude of their P&B effects. Second, rankings of individual tuning sensitive queries are combined to estimate the overall ranking of a parameter for the workload. The queries which are insensitive to parameter tuning are not included in the workload ranking calculation. Therefore, we do not consider query Q3.

To rank the configuration parameters for a tuning sensitive query, the estimated P&B effects are normalized with respect to the maximum effect and the range of normalized effects are divided into N buckets, where N is number of configuration parameters. A parameter is assigned to the rank i if its normalized effect falls into the i-th bucket range. In the continuing example, we have seven parameters. Therefore the range for the first, second, third, fourth, fifth, sixth, and seventh buckets are, [1, 0.86), [0.86, 0.71), [0.71, 0.57), [0.57, 0.43), [0.43, 0.29), [0.29, 0.14), and [0.14, 0.0], respectively. The normalized and rounded P&B effects for the queries Q1 and Q2 are listed in Table 3. For Q1, the rank of P_1 is 3 as its normalized P&B effect 0.56 falls into the third bucket. Similarly, the rank of P_2 is 4, and so on. The ranking of parameters for queries Q1 and Q2 are listed in Table 4. In the next step, ranks are summed across all tuning sensitive queries, averaged, and sorted in ascending order. The most important parameters will have the lowest cumulative rank. The average rankings of the parameters P_1, P_2, P_3, P_4, P_5, P_6 and P_7 of queries Q1 and Q2 as listed in Table 4 are 4.0, 2.5, 4.0, 1.5, 6.5, 4.0, and 2.0, respectively. Therefore, final ranking is 4, 3, 4, 1, 7, 4, and 2, respectively. Ranking indicates that P_4 is the most important configuration parameter, P_7 is the second most important configuration parameter, followed by P_2, $\{P_1, P_3, P_6\}$, and P_5, in order. A detailed description of this methodology can be found in [10].

4.2 Compressing a Query Workload

To select a compressed query, all queries of the original workload are divided into two groups: tuning sensitive and insensitive. One query from the insensitive group is included in the compressed query workload, while the tuning sensitive group is further divided into subgroups based on the similarities of the effects of the configuration parameters. For each query, the effects are normalized to the maximum effect of the parameters for the corresponding query. Then, the Euclidean distance of the normalized effects among the queries is calculated to estimate a similarity score. If the Euclidean distance between the effects of the two queries is less than the user-chosen similarity threshold, we consider them as similar queries in terms of the performance bottlenecks and place them in the same group. Finally, one query is selected from each subgroup to include in the compressed query workload.

In the continuing example, the tuning sensitive group consists of queries Q1 and Q2; and the insensitive group consists of query Q3. The Euclidean distance between the normalized effects of queries Q1 and Q2 from Table 3 is $\sqrt{(0.56-0.33)^2+(0.41-0.87)^2+\ldots+(0.13-1.0)^2+(0.92-0.59)^2} = 1.36$. If the threshold of similarity score is 1.50, then we can consider queries Q1 and Q2 to be similar in terms of their impact of the performance bottleneck parameters. In the compressed query workload, we can include either query Q1 or query Q2. If we select query Q1, then the compressed query workload consists of queries Q1 and Q3. A detailed description of this methodology can be found in [21].

5 Experimental Results

All the experiments in this section are conducted in a machine with two Intel XEON 2.0 GHz w/HT CPUs, 2 GB RAM, and 74 GB 10,000 RPM disk. We use the TPC-H benchmark [2] and PostgreSQL [1] to demonstrate our methodologies. For demonstration, we use 22 read-only TPC-H queries, Q1-Q22, and data size of 1 GB. We consider only those PostgreSQL configuration parameters that are relevant to the read-only queries. The high and low values of each parameter are chosen in a range such that it will act as a monotonic parameter. A detailed information of the values used can be found in [21]. Furthermore, we have included some parameters, for example, `fsync` and `checkpoint_timeout`, which are not relevant for the read-only queries, yet they help in verifying that our method is working correctly. If their rankings or effects become low compared to other parameters, then it will give an indication that our method correctly identifies the performance bottlenecks.

The P&B effects of all configuration parameters are calculated using the *P&B design with foldover*. In order to identify the insensitive queries, COV of 0.05 is selected as a threshold. Out of 22 queries, the COVs of the queries Q4, Q6, Q7, Q10, Q11, Q12, Q14, Q15, Q17, Q18, and Q22 are found to be less than 0.05. These 11 queries formed one single group of tuning insensitive queries. The ranking of the parameters for tuning sensitive queries is listed in Table 5. For more detailed results of the estimated and normalized P&B effects, the readers are referred to [21]. Different parameters with the same rank indicates that they have similar effects on the performance. For query Q1, `work_mem` is the most important parameter, while other parameters do not have any impact on performance. For query Q2, `effective_cache_size` and `shared_buffers` are the first and sixth most important parameters, respectively, while other parameters do not have significant impact on performance. Similarly, the ranking of the parameters for other queries indicates the relative importance of corresponding parameter in the query performance.

The ranking of the parameters for the original TPC-H query workload consisting of tuning sensitive queries is listed in the first and second columns of Table 6. The results indicate that `work_mem` is the most important configuration parameter, `shared_buffers` and `effective_cache_size` are second most important parameters, followed by `cpu_operator_cost`, `random_page_cost`, and so on. The result also indicates that `fsync` and `checkpoint_time_out` do not appear in the top five most important parameters list. To verify that our results match with the decisions made by DBAs, we compare our parameter ranking against the PostgreSQL 8.0 Performance Checklist [5]. This checklist is a set of rules of thumb for setting up PostgreSQL server where it suggests the settings of configuration parameters that most DBAs will want to change. Among the parameters we are considering, according to this checklist there are six important parameters that need to be tuned, namely, `max_connections`, `shared_buffers`, `work_mem`, `maintenance_work_mem`, `effective_cache_size`, and `random_page_cost`. Four of these six parameters appear in our top five important parameters list. The differences between our result and this list are: (1) we find that the parameter `cpu_operator_cost` is an important one to our query workload and (2) the parameter `max_connections` appears to be less important to our workload as we do not consider concurrently running queries. Therefore, in general, our ranking methodology matches the general guidelines that are suggested for database tuning in addition to adding specific tuning decisions that match the given query workload.

To select a compressed query workload, we set a threshold of 0.5 for the Euclidean distance. At this threshold the 11 tuning sensitive queries form eight groups: {Q1, Q8, Q16}, {Q2, Q13}, {Q3}, {Q5}, {Q9}, {Q19, Q21}, and {Q20}. In the compressed workload, from each group we include the query which creates less perturbations in the original query workload ranking. In addition, we have to include one query from the insensitive group in the compressed workload. We select the query which has the largest execution time. Our compressed query workload includes queries Q2, Q3, Q5, Q8, Q9, Q18, Q20, and Q21. The new ranking for the compressed query workload is given in the third and fourth columns of Table 6. The result indicates that except `shared_buffers` and `effective_cache_size`, the ranking of the rest of the parameters is identical. `shared_buffers` is ranked second in the original query workload, while it is ranked third in the compressed query workload. On the other hand, `effective_cache_size` is ranked second in the original query workload, while it is ranked

Parameter	**Q1**	**Q2**	**Q3**	**Q5**	**Q8**	**Q9**	**Q13**	**Q16**	**Q19**	**Q20**	**Q21**
checkpoint_timeout	15	15	15	13	15	4	15	15	15	7	15
deadlock_timeout	15	15	13	15	13	4	15	15	15	7	14
fsync	15	15	15	15	15	2	15	15	15	6	15
max_connections	15	15	12	13	13	11	15	15	15	6	14
shared_buffers	15	6	15	15	15	12	7	14	1	6	1
stats_start_collector	15	15	15	15	15	11	15	15	15	7	14
cpu_index_tuple_cost	15	15	13	15	15	9	15	15	15	7	12
cpu_operator_cost	15	15	10	13	15	1	15	15	15	1	15
cpu_tuple_cost	15	15	13	13	14	8	15	15	15	7	14
effective_cache_size	15	1	11	13	15	1	1	15	15	6	13
geqo	15	15	9	13	12	10	15	15	15	7	14
maintenance_work_mem	15	15	11	15	11	10	15	15	15	6	12
random_page_cost	15	15	13	1	13	4	15	15	15	12	15
temp_buffers	15	15	11	15	15	3	15	15	15	7	15
work_mem	1	15	1	13	1	9	15	1	15	7	15

Table 5: Ranking of the configuration parameters for the tuning sensitive TPC-H queries.

Rank	**Original Workload**	**Rank**	**Compressed Workload**
1	work_mem	1	work_mem
2	effective_cache_size	1	effective_cache_size
2	shared_buffers	3	shared_buffers
4	cpu_operator_cost	3	cpu_operator_cost
5	random_page_cost	5	random_page_cost
6	geqo	6	geqo
6	maintenance_work_mem	6	maintenance_work_mem
6	deadlock_timeout	6	deadlock_timeout
6	temp_buffers	6	temp_buffers
10	max_connections	10	max_connections
10	cpu_tuple_cost	10	cpu_tuple_cost
10	fsync	10	fsync
10	checkpoint_timeout	10	checkpoint_timeout
14	cpu_index_tuple_cost	14	cpu_index_tuple_cost
15	stats_start_collector	15	stats_start_collector

Table 6: Ranking of the configuration parameters estimated by the original and compressed query workloads.

first in the compressed query workload. However, as long as the list of topmost important parameters does not change drastically, in reality it does not cause much impact in tuning activities.

6 Related Work

Major database vendors offer tools for tuning database physical design [3, 9, 22]. IBM DB2 provides a utility named `autoconfigure` for automatically selecting the initial values for the configuration parameters based on generic workload behavior [15]. Oracle Automatic Database Diagnostic Monitor (ADDM) tool possesses a holistic view of the database, identifies root causes of the performance bottlenecks, and estimates the benefits of eliminating performance bottlenecks [12]. In Microsoft SQL Server, most of the parameters can be configured either through Enterprise Manager or with the T-SQL `sp_configure` command [11]. However, none of the current tools rank the configuration parameters based on their impact on the DBMS performance.

Two major techniques for query workload compression are proposed in the literature. The first technique groups SQL statements based on the accessed tables and join columns [7]. The second technique focuses on the most complex and costly queries in the workload and ignore other queries [22]. In contrast, our proposed workload compression methodology selects subset workload based on similarities of the performance bottlenecks of the configuration parameters.

7 Conclusion

We have proposed methodologies for ranking configuration parameters and selecting a compressed query workload based on the impact of configuration parameters on the DBMS performance for a given query workload. These methodologies are quite generic and can also be applied to non-database systems. They will greatly help DBAs of all knowledge levels to prioritize tuning activities and reduce time and effort. In the future, we are planning to perform the following extensions: 1) validating the assumptions behind calculating parameter effects, and 2) suggesting the appropriate values of the configuration parameters using the estimated P&B effects.

8 Acknowledgements

This work was supported in part by NSF grant nos. CCF-0621462 and CCF-0541162, the University of Minnesota Digital Technology Center Intelligent Storage Consortium, and the Minnesota Supercomputing Institute.

References

[1] PostgreSQL DBMS Documentation. `http://www.postgresql.org/`.

[2] Transaction Processing Council. `http://www.tpc.org/`.

[3] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proc. of VLDB*, 2004.

[4] T. Allen. *Introduction to Engineering Statistics and Six Sigma: Statistical Quality Control and Design of Experiments and Systems*. Springer, 2006.

[5] J. Berkus. Power PostgreSQL: PostgreSQL Performance Pontificated. `http://www.powerpostgresql.com/PerfList/`.

[6] D. Cappucio, B. Keyworth and W. Kirwin. The Total Cost of Ownership: The Impact of System Management Tools. Strategic Analysis Technical Report, Gartner Group, Stamford, CT, 1996.

[7] S. Chaudhuri, A. K. Gupta and V. Narasayya. Compressing sql workloads. In *Proc. of SIGMOD*, 2002.

[8] S. Chaudhuri and G. Weikum. Rethinking Database Architecture: Towards s Self-tuning RISC-style Database System. In *Proc. of VLDB*, 2000.

[9] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proc. of VLDB*, 2004.

[10] B. Debnath, D. Lilja and M. Mokbel. SARD: A Statistical Approach for Ranking Database Tuning Parameters. In *Proc. of 3rd Intl. Workshop on Self-Managing Database Systems*, 2008.

[11] S. DeLuca, M. Garcia, J. Reding and E. Whalen. *Microsoft SQL Server 7.0 Performance Tuning Technical Reference*. Microsoft Press, March 2000.

[12] K. Dias, M. Ramacher, U. Shaft, V. Venkataramamani and G. Wood. Automatic Performance Diagnosis and Tuning in Oracle. In *Proc. of CIDR*, 2005.

[13] C. Garry. Who's Afraid of Self-Managing Databases? `http://www.eweek.com/article2/0,1895,1833662,00.asp`, June 30, 2005.

[14] Hurwitz Group. Achieving Faster Time-to-Benefit and Reduced TCO with Oracle Certified Configurations. March, 2002.

[15] E. Kwan, S. Lightstone, A. Storm and L. Wu. Automatic Configuration for IBM DB2 Universal Database. In *IBM Perfromance Technical Report*, January 2002.

[16] S. Lightstone, G. Lohman, P. Haas, V. Markl, J. Rao, A. Storm, M. Surendra and D. Zilio. Making DB2 Products Self-Managing: Strategies and Experiences. *IEEE Data Engineering Bulletin*, 29(3), 2006.

[17] D. Lilja. *Measuring Computer Performance A Practitioner‘s Guide*. Cambridge University Press, 2000.

[18] D. Montgomery. *Design and Analysis of Experiments*. Wiley, 2001.

[19] R. Plackett and J. Burman. The Design of Optimum Multifactorial Experiments. In *Biometrika Vol. 33 No. 4*, 1946.

[20] A. Rosenberg. Improving Query Performance in Data Warehouses. `http://www.tdwi.org/Publications/BIJournal/display.aspx?ID=7891`, 2005.

[21] J. Skarie, B. Debnath, D. Lilja and M. Mokbel. SCRAP: A Statistical Approach for Creating Compact Representational Query Workload based on Performance Bottlenecks. In *Proc. of IISWC*, 2007.

[22] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano and S. Fadden. DB2 Design Advisor: Integrated Automated Physical Database Design. In *Proc. of VLDB*, 2004.

Automated Workload Management for Enterprise Data Warehouses

Abhay Mehta Chetan Gupta Song Wang Umeshwar Dayal

Hewlett Packard Labs
firstname.lastname@hp.com

Abstract

Modern enterprise data warehouses have complex workloads that are notoriously difficult to manage. Additionally, RDBMSs have many "knobs" for managing workloads efficiently. These knobs affect the performance of query workloads in complex interrelated ways and require expert manual attention to change. It often takes a long time for a performance expert to get enough experience with a large warehouse to be able to set the knobs optimally. Typically the warehouse and its workload change significantly within that time. This makes the task of manually optimizing the knob settings on a warehouse an impossible one. In this context, our goal is to create self managing Enterprise Data Warehouses. In this paper we describe some recent advances in building an automatic workload management system. We test this system against real workloads against real enterprise data warehouses.

1 Introduction

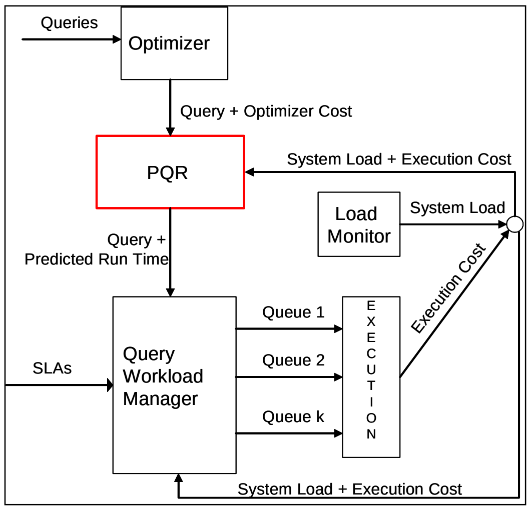
Many organizations are creating and deploying Enterprise Data Warehouses (EDW) to serve as the single source of corporate data for business intelligence. Not only are these enterprise data warehouses expected to scale to enormous data volumes (hundreds of terabytes), but they are also expected to perform well under increasingly complex workloads, consisting of batch and incremental data loads, batch reports and complex ad hoc queries.

The problem of database workload management aimed at self-tuning database systems has been studied in the literature (see Weikum [28] for a review of the advances in this area). We have borrowed from this work the idea of using multiprogramming level (MPL) to model the load on the system. However, the previous work was done in the context of OLTP workloads, not the complex query workloads typical of Business Intelligence (BI) data warehouses, which is the focus of our work.

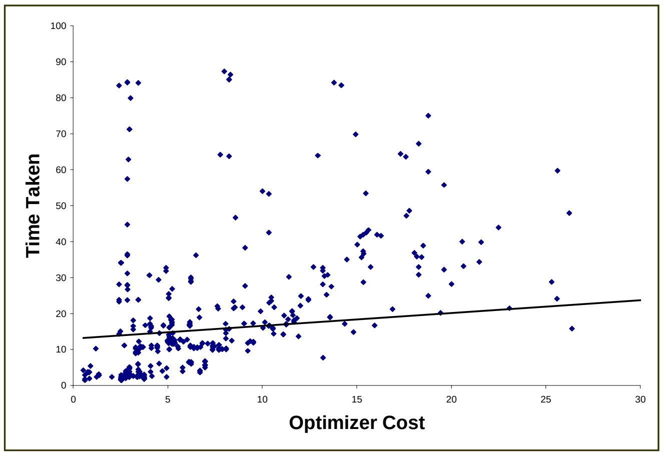
In this work we deal with two important challenges towards achieving Automatic Workload Management: *Predictability* and *Manageability*. In the next few sections, for each of these challenges we present a sketch of our solution. The detailed discussions and results can be found in [16] and [17].

Figure 1(a) depicts the architecture of an automatic workload management system. The *optimizer* outputs an execution plan for a query and an estimate of the query cost, which are input to the *Prediction of Query Runtime (PQR)* block. In addition, a *Load Monitor* extracts a load feature vector, which is also input to the *PQR* block.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a)



(b)

Figure 1: (a) System Diagram for an Autonomic EDW (b) Relationship between Optimizer Cost and Actual Execution Time

Whenever a new query comes in, the *PQR* block estimates the execution time of the query under current load conditions. This estimate is passed on to the *Workload Manager*, which schedules the queries. Other components in the system (not shown in the figure) keep track of the query's progress relative to its predicted execution time, and use this information to detect problem queries (such as runaway queries). All of this information is fed back to the *Workload Manager*, which can then apply the appropriate control actions to rectify the problems.

2 Related Work

There has been a tremendous amount of work on cost models for query optimization (see for example Graefe [9] for a survey). However, while these cost models are useful to the optimizer for selecting low cost execution plans, their cost estimates are very often not good predictors of actual query execution times (See Figure 1(b)).

Analytical approaches have been used for estimating query response times [23, 27] and there are a few commercial products that use analytical and simulation models to predict query execution times [1,14,20,21,25]. The analytical approaches depend on the creation of resource models which are notoriously complex and difficult to create and hence the results may not be relevant in practice.

Certain machine learning techniques have been used in the context of databases. The LEO learning optimizer uses a feedback loop of query statistics to improve the optimizer during run time [15, 26]. Raatikainen [22] summarizes some of the early work in using clustering for workload classification. In PLASTIC [8] queries are clustered to increase the possibility of plan reuse. Although these are interesting applications of machine learning techniques, none of these apply machine learning to our problem at hand.

Furthermore, statistical techniques, analytical techniques, and machine learning techniques have been used previously to predict execution times of tasks and resource consumption in fields other than database systems [10]. However, to the best of our knowledge there is no prior work in using machine learning techniques to build models for predicting query execution time ranges.

The related work for throughput control falls into three areas: thrashing control in operating systems, creative memory management in DBMSs, feedback control of workloads.

The problem of over subscription of memory, the primary cause of thrashing has been studied extensively since the 1960s. Several heuristics have been proposed for doing admission control either by explicitly controlling the MPL or otherwise. These include the Knee Criterion, the L=S criterion, the Page Fault Frequency algorithm, and the 50% rule [6]. However, thrashing is still an unsolved problem in operating systems and work continues in this area [11].

Another area of related work is in the design of memory managers for DBMSs. Several proposals have been made for memory managers in DBMS: [2–4]. The drawback of these methods is that the internal workings of the database memory manager have to be changed.

One more area of related work is in the feedback control of workloads. Most of the previous work using this approach has been targeted towards OLTP (On-line Transaction Processing) systems where thrashing due to data contention has been the main problem. Several of these methods have been summarized in [18]. Another good demonstration of this approach is provided by [19] that deals with real-time database systems, and by [24]. More recently, Web servers have employed a feedback loop approach [5, 7, 12, 13].

To our knowledge, the most common approach used by commercial BI systems is a "static MPL" approach. In this approach, a "typical workload" is run multiple times through the system and an appropriate MPL computed. The workload is then "throttled" down to this static MPL, which may be different for different times of the day. There are several problems with this approach: it is expensive, it results in a very approximate and inaccurate setting and since it is static it is not suitable for heterogeneous nature of a BI workload.

3 Predictability

The execution time of BI queries that run on large EDWs can vary from microseconds for simple lookup queries all the way to multiple hours for complex data mining queries. An effective workload management system depends heavily on an estimate of the execution times of queries in the workload, prior to running the queries. Estimating execution time accurately is a hard problem, especially on a loaded EDW with a complex workload.

Previous researchers have focused on predicting precise execution times. In our experience, this is extremely difficult to do with high accuracy. Furthermore, for workload management, it is actually unnecessary to estimate a precise value for execution time - it is sufficient to produce an estimate of the query execution times in the form of time ranges (for instance, queries may be assigned to different queues based on their execution time ranges). This allows us to reformulate the problem and bring in the machinery of machine learning to address it. It is precisely this problem of estimating query execution time ranges that we address in this paper. We focus on the following issues:

1. Discovering, selecting, and computing query plan features and system load features as classification attributes.

2. Finding appropriate execution time ranges to be used for prediction.

3. Ensuring high prediction accuracy.

4. Efficient algorithms for model building and deployment.

Researchers and practitioners have built increasingly sophisticated cost models for query optimization. However, building an accurate analytical model is difficult especially under varying load conditions. Using an optimizer's analytical cost model to estimate the actual execution time of a query on a loaded system has met with limited success in the field and it is common knowledge that query cost estimates produced by query optimizers do not accurately reflect query run times. For example, in Figure 1(b), we have plotted the actual query execution time and cost estimates for a batch of queries run on a customer database. The scatter plot and the best fit line show that the optimizer cost is not an ideal predictor for query execution time.

We take a different approach: we "learn" from the execution histories of various queries under varying load conditions. In particular, from the execution histories, we extract query plan features provided by the optimizer and system load features from the environment on which the queries were run. Discovering features and selecting which features to use is itself a challenging problem. We then build a predictive model that can estimate the execution time range of a query.

We found that conventional machine learning approaches to building predictive models, such as regression and decision tree classifiers were not adequate. The challenges were several since we are interested in not only predicting the time ranges but also in discovering them:

1. The time ranges should be sufficient in number. It would be meaningless to predict that all queries belong to a single time range.

2. Their span should be meaningful. Very small or very large time buckets are not very useful.

3. As with all predictive models the accuracy of prediction should be high.

4. The model should be cheap to build and deploy.

To address these we came up with a novel hierarchical approach. We call the predictive models built using this approach, PQR (Predicting Query Run-time) Trees. A PQR Tree is a hierarchical classification tree such that for every node, there is a binary classifier that decides how best to divide the time range of the node into two sub ranges for the two children of the node. There is also an associated accuracy for each node. Every node and leaf of the tree corresponds to a time range. At every node, we not only find the two sub ranges for the time range of the node but also a classifier that can predict the two ranges.

As illustrated in Figure 2(a), the prediction model is first built and trained using a set of execution histories of various queries under varying load conditions. Then, for an incoming query in the workload, PQR Tree will return an estimated execution time range with an associated accuracy. There are two overall steps:

1. *Obtaining a PQR Tree based on historical data of queries run on the system*: This involves three steps:

   (a) From the historical data extract query plan features like optimizer cost, number of joins, join cardinality, and etc.

   (b) From the historical data extract the system load vector for each query. The system load vector consists of the number of queries and the number of processes running while the query was executing.

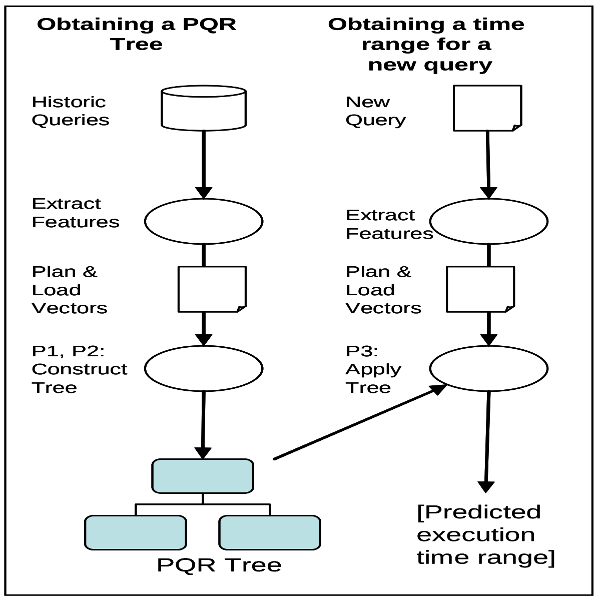
   (c) Build a PQR Tree with the feature vectors built above. This is done by choosing the combination of the classifier and the time interval that gives the highest accuracy.

2. *Obtaining a time range for a new query by applying the PQR Tree to a new query*: This is done by extracting a feature vector from the new query and applying the PQR Tree obtained in the previous step.

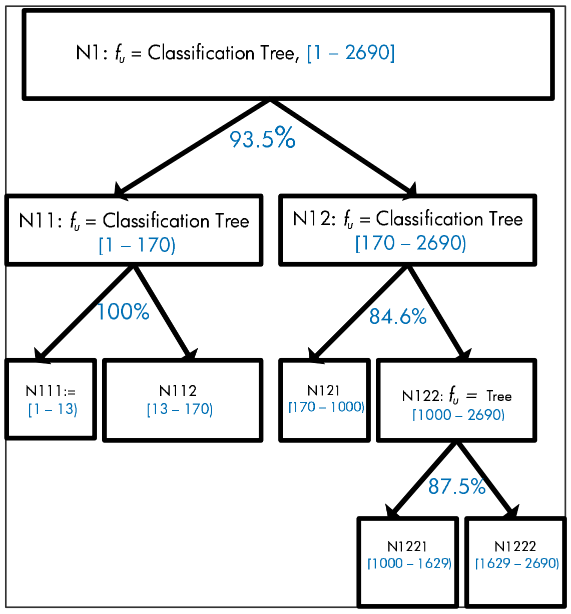
A sample PQR Tree is presented in Figure 2(b). The classifier, f_u associated with the root node divides the time range of [1, 2690] seconds into two: [1, 170) and [170, 2690] seconds with associated accuracy of 93.5%. The rest of the nodes can be interpreted similarly.

We did two series of experiments to verify our approach. They consisted of two different systems, both running a commercial, enterprise class DBMS and two different data sets. We ran a thousand queries against SF = 1, 50, 100, 200 TPCH databases. The queries were generated automatically and ran against all the tables. They include joins and order-bys. Sixteen of these queries were TPCH benchmark queries. We looked at four query MPL values: 4, 6, 8 and 10, i.e., we executed the queries in parallel with the number of parallel streams equal to the MPL value.

For the second set we took a set of actual BI (Business Intelligence) queries run in a day by one of HP's customers. They include both ad hoc queries and canned reports. There were a total of 500 queries in this data set. The database has more than 38G rows and total size is over 21TB. This experimental setup consisted of



(a)


N1: f_u = Classification Tree, [1 – 2690]
93.5%
N11: f_u = Classification Tree [1 – 170]
N12: f_u = Classification Tree [170 – 2690]
100%
84.6%
N111:= [1 – 13]
N112 [13 – 170]
N121 [170 – 1000]
N122: f_u = Tree [1000 – 2690]
87.5%
N1221 [1000 – 1629]
N1222 [1629 – 2690]

(b)

Figure 2: (a) Solution Components for Building PQR Tree (b) Sample PQR Tree where Time is in Seconds

hundred different tests. We took ten different MPLs: 8, 16, 32, 48, 64, 96, 128, 192, 224, 256 and ran each experiment ten times. In another series of hundred different tests we used a subset of smaller queries from the original five hundred queries.

In consultations with DBAs who manage workloads, we created a quality metric: a model that can predict at least four "reasonable" ranges[1] of query execution times with an accuracy of greater than 80% is considered acceptable. This metric captures all the key attributes we discussed earlier. Over 90% of the PQR Tree models were found to be acceptable. The results from various experiments were found to be extremely encouraging [16].

4 Manageability

Traditionally, the Multi Programming Level (MPL), which indicates the number of queries running concurrently on the system, has been used to control the load on the system. To avoid system underload or overload, MPL must be carefully set. Figure 3(a) shows the throughput curves for three different TPC-H workloads. Each workload has a range of MPL values for which there is no overload or underload. Clearly, different workloads require different optimal MPLs. However, a typical BI batch workload can fluctuate rapidly between long, resource intensive queries, and short, less intensive queries. This requirement makes it very challenging, if not impossible for a human (or a system) to keep the workload in an optimal region using the MPL setting.

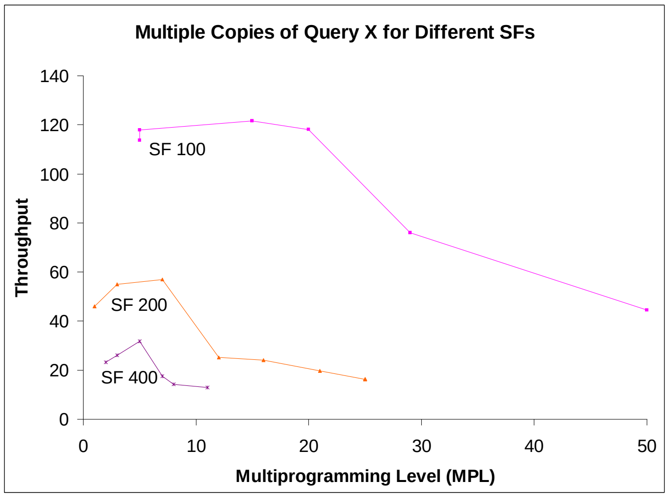
We have created a BI Batch Manager, which is a database workload management system for running batches of BI queries. The BI Batch Manager has the following salient points:

1. It employs new way of executing BI queries, Priority Gradient Multiprogramming (PGM), which automatically protects against overload.

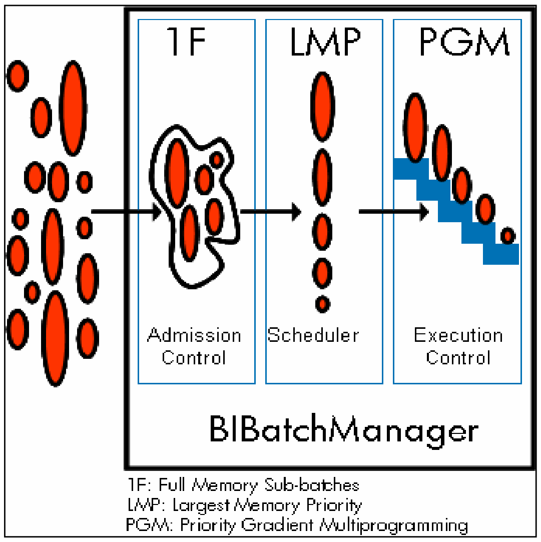
2. The scheduling algorithm, Largest Memory Priority (LMP) further enhances the stability of execution.

3. Estimated memory is used as a basis for admission control in EDWs.

[1]We ensure "reasonableness" by stipulating that no child node has less than 25% examples of the parent node.



(a)


1F
LMP
PGM
Admission Control
Scheduler
Execution Control
BIBatchManager
1F: Full Memory Sub-batches
LMP: Largest Memory Priority
PGM: Priority Gradient Multiprogramming

(b)

Figure 3: (a) Throughput as a Function of MPL (b) Component Design for BI Batch Manager

A common way of looking at throughput is by means of throughput curves where the throughput is plotted against the MPL on the system. When a user first confronts a new workload the precise shape of the throughput curve is unknown to him/her and the user has to determine the MPL at which to execute the workload. The user does not want to be on the left part of the curve since increasing the MPL can lead to an increase in throughput. But as the MPL is increased there is a danger of entering the overload region where higher MPLs mean a significantly lower throughput. At the boundary between the optimal region and the overload region, increasing the MPL by even one, can cause severe performance deterioration rather than a gradual decline in performance. This is because of thrashing, a problem that is inherent in all virtual memory, multiprogramming systems.

Our focus is on addressing the problem of automatically managing BI batch workloads, so that we are in the optimal region of the throughput curve, where there is no underload or overload. We focus on issues as follows:

1. Identify a manipulated variable whose predicted value is suitable for BI workload management.
2. Make the execution of the queries stable over a wide range of estimation errors of this variable.
3. Schedule the queries so that the system behaves without underload or overload for the admitted batch.
4. Use the manipulated variable for admission control, i.e. admit queries based on an estimated value of the manipulated variable.

Our solution is depicted in Figure 3(b). The BI Batch Manager has three primary components, that address the issues highlighted above: (1) *Admission Controller*, (2) *Scheduler* and (3) *Execution Manager*.

We use memory as the variable of choice to manipulate. Whenever there is an over-subscription of memory, there is a potential for serious degradation in performance due to thrashing. A workload thrashes whenever its cumulative peak memory requirement per CPU exceeds the available memory per CPU. Overload behavior can be predicted more accurately with memory rather than with MPL. Thus, in contrast to MPL, memory behaves much more predictably as a manipulated variable.

Current execution control technology centers around Equal Priority Multiprogramming (EPM), in which every query is executed at the same process priority. EPM is robust for a reasonable range of overestimates of the memory requirement. However, it is very unstable for underestimates, which will result in a sudden drop in throughput as the size of the workload memory increases beyond the available memory at runtime.

To overcome the sensitivity to thrashing for underestimates of memory[2], we introduce Priority Gradient Multiprogramming (PGM). In PGM, queries are executed at different process priorities such that a gradient of priorities is created. PGM requires that the operating system supports preemptive priority scheduling, which is standard on many commercial systems, including Linux and NSK. This results in queries asking for, and releasing resources at different rates. This solution has proven to be very effective in protecting against overload. Ironically, PGM is an effective execution control mechanism because it uses the priority gradient to distribute memory to queries in an unfair way. The priority gradient allows the operating system to automatically allocate resources to queries down the gradient without any over-allocation of resources and keeps the system in an optimal range of execution.

Since the throughput penalty for being in the overload region is much higher than being in the underload region, we designed a scheduler that stabilizes the system for memory underestimation errors. We call our ordering scheme: Largest Memory Priority (LMP). The query with the largest memory requirement is given the highest priority.

For admission control we create batches whose estimated memory requirement is equal to the available memory per CPU on the system. The whole batch is divided into these sub-batches using the standard technique of bin packing called First Fit Decreasing(FFD). Once a batch finishes, a new batch is admitted for execution. Here, we can have various definitions of what is meant by a batch being "done". For example, one definition could be that a batch is done when 90% of the queries in the batch are done or if all the memory is released. Estimation errors are compensated for, by our PGM execution control mechanism as described previously.

Our experiments have shown that the BI Batch Manager (BIBM) does not cause thrashing for memory estimates that underestimate the memory requirement to be a third of what it actually is. Similarly, it does not go into underload if we overestimate the memory to upto three times the actual memory requirement. Also, most DBMS put bounds on the memory available to BMOs. This reduces the extent to which memory can be underestimated. Finally, in our paper [17] we give a statistical proof that shows that errors in memory estimates of a batch are much less than the errors in the memory estimates of the individual queries. Thus, the main contribution of BIBM is that it makes the system tolerant of memory estimation errors. BIBM compensates upto a factor of 3, or 300% error in the estimate for how much memory a workload is going to need. This is seen as a sufficient margin of error for most practical workloads.

We have done a series of experiments to test various aspects of our BI Batch Manager. For the purpose of experimentation we used a TPC-H workload with three SF values: 50, 100, 200. It was installed on a two Segment (32 Nodes) commercial class Enterprise Data Warehouse, with 8GB physical memory per CPU. We created forty-eight mixed workloads of random sizes by uniform random sampling (with replacement).

We compared the throughput obtained with BIBM with the *EPM Throughput* (throughput achieved when all queries are running with the same priority) and the *Ideal Throughput* (theoretic maximum throughput achieved when CPU utilization is 100%). Note that, in practice it is impossible to obtain the ideal throughput, since even for a highly parallelized query there are a number of serial operations. Thus, the ideal throughput should be viewed as a good upper bound, but not necessarily achievable.

In our experiment result shown in Figure 4, BI Batch Manager generally achieved a system throughput of greater than 80% of ideal throughput. There was approximately 4 GB of memory available per CPU during the experimental runs. The workloads were created by first choosing a memory number between 1.33GB and 12GB (approximately 1/3 to 3 times of memory per CPU). Then queries were randomly chosen from TPC-H queries (with replacement) until the memory boundary was reached. More experiment results are available in [17].

[2]If memory is under-estimated, a batch that requires larger amount of memory might be submitted causing thrashing.

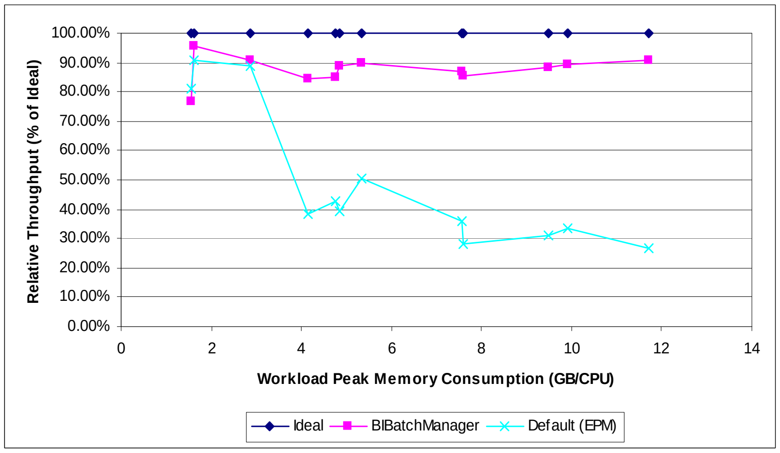


Figure 4: Throughput Results for Workloads from TPC-H SF200

5 Conclusion

In this work we have dealt with predictability and manageability in an effective way. Our results have been validated on real life commercial class EDWs. As a next step we plan to extend manageability to a continuous stream of queries and detect problem queries.

References

[1] BEZ Systems Inc. BEZPlus for NCR Teradata and Oracle environments on MPP machines. http://www.bez.com/software.htm, 1999.

[2] K. P. Brown, M. J. Carey and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB*, pages 328–341, 1993.

[3] K. P. Brown, M. Mehta, M. J. Carey and M. Livny. Towards Automated Performance Tuning for Complex Workloads. In *VLDB*, pages 72–84, 1994.

[4] M. J. Carey, R. Jauhari and M. Livny. Priority in DBMS resource scheduling. In *VLDB*, pages 397–410, 1989.

[5] X. Chen, P. Mohapatra and H. Chen. An admission control scheme for predictable server response time for web accesses. In *WWW*, pages 545–554, 2001.

[6] P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier and R. Suri. Optimal Multiprogramming. *Acta Inf.*, 7:197–216, 1976.

[7] S. Elnikety, E. M. Nahum, J. M. Tracey and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW*, pages 276–286, 2004.

[8] A. Ghosh, J. Parikh, V. S. Sengar and J. R. Haritsa. Plan Selection Based on Query Clustering. In *VLDB*, pages 179–190, 2002.

[9] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[10] M. A. Iverson, F. Özgüner and G. J. Follen. Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. In *HPDC*, pages 263–270, 1996.

[11] S. Jiang and X. Zhang. TPF: a dynamic system thrashing protection facility. *Softw., Pract. Exper.*, 32(3):295–318, 2002.

[12] A. Kamra, V. Misra and E. M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites. In *IWQoS*, pages 47–56, 2004.

[13] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein and S. S. Parekh. Online Response Time Optimization of Apache Web Server. In *IWQoS*, pages 461–478, 2003.

[14] M. Garth. Modelling parallel architectures. http://www.metron.co.uk/papers.htm, 1996. Metron Technology white paper.

[15] V. Markl, G. M. Lohman and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[16] A. Mehta, C. Gupta and U. Dayal. How to Predict the Running Time of Business Intelligence Queries on an Enterprise Data Warehouse. Technical Report HPL-2007-165, HP Labs, October 2007.

[17] A. Mehta, C. Gupta and U. Dayal. BI Batch Manager: A System for Managing Batch Workloads on Enterprise Data Warehouses. In *EDBT (To appear)*, 2008.

[18] A. Mönkeberg and G. Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. In *VLDB*, pages 432–443, 1992.

[19] H. Pang, M. J. Carey and M. Livny. Managing Memory for Real-Time Queries. In *SIGMOD*, pages 221–232, 1994.

[20] Platinum Technology. Proactive performance engineering. http://www.softool.com/products/ppewhite.htm, 1999. Platinum Technology White paper.

[21] R. Eberhard. DB2 Estimator for Windows. http://www.software.ibm.com/data/db2/os390/estimate, 1999. IBM Corp.

[22] K. E. E. Raatikainen. Cluster Analysis and Workload Classification. *SIGMETRICS Performance Evaluation Review*, 20(4):24–30, 1993.

[23] S. Salza and M. Renzetti. Performance Modeling of Paralled Database System. *Informatica (Slovenia)*, 22(2), 1998.

[24] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. M. Nahum and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *ICDE*, page 60, 2006.

[25] SES Inc. Solutions for information systems performance. http://www.ses.com/Solution/IS.html, 1999.

[26] M. Stillger, G. M. Lohman, V. Markl and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[27] N. Tomov, E. W. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. B. King and P. Broughton. Analytical response time estimation in parallel relational database systems. *Parallel Computing*, 30(2):249–283, 2004.

[28] G. Weikum, A. Mönkeberg, C. Hasse and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*, pages 20–31, 2002.

Quality of Service-Enabled Management of Database Workloads

Stefan Krompass[‡] Andreas Scholz[‡] Martina-Cezara Albutiu[‡]
Harumi Kuno[§] Janet Wiener[§] Umeshwar Dayal[§] Alfons Kemper[‡]

[‡]Technische Universität München, Munich, Germany
{albutiu,kemper,krompass,scholza}@in.tum.de

[§]Hewlett-Packard Laboratories, Palo Alto, CA, USA
{firstname.lastname}@hp.com

Abstract

Database administrators struggle when managing workloads that have widely different performance requirements. For example, the same database may support short-running OLTP queries and batch jobs containing multitudes of queries with varying complexity. Different workloads may have different performance requirements, expressed in terms of service level objectives (SLOs) that must be fulfilled in order to keep the issuing database users satisfied. In this paper, we identify basic query classes and describe the challenges they pose for SLO-aware workload management. Additionally, we propose a generic architecture for an SLO-aware DBMS. We give an overview of workload management techniques already implemented in today's DBMS and outline future research directions for as yet unsupported concepts.

1 Introduction

Imagine you are a database system administrator for a large company. Your job is to administer a variety of workloads running on the database. These workloads are submitted by different customers who have unique requirements. The company's Web front end produces an OLTP-style workload with short-running parameterized queries that must be processed quickly in order to provide immediate feedback to the customers. Depending on the customer, the queries have varying importance and performance requirements. While processing the OLTP queries, your database must also handle business intelligence (BI) workloads. For example, sales managers submit analytic batch workloads to prepare financial reports for a meeting with your company's CEO. These workloads have a hard deadline and partial results are worthless. To complicate matters, members of the marketing team simultaneously need to execute complex custom queries in order to craft their new campaign.

In today's databases, OLTP and BI workloads are usually kept separate: OLTP workloads are submitted to and processed by operational databases, and BI workloads by data warehouses. For the management of each individual workload, you as the database administrator must address a variety of problems: First, you need concrete metrics that describe the customers' expectations in a way you can measure. For example, you cannot measure whether or not you meet the customer's vague expectation of a "short response time", but you can measure the elapsed time needed to respond to a query. Second, you need policies to manage incoming queries. For example, you must decide whether or not to admit a new query when you expect the query to have a negative impact on concurrent queries. Third, you need workload management policies that consider the characteristics of the workloads as a whole. In particular, you need workload management techniques to address questions like:

- What is the number of concurrent queries in the database system that optimizes throughput for a particular workload? What if multiple workloads are running simultaneously?
- Can the queries be scheduled according to their priorities? How should the priorities be determined?
- How long should you wait before killing an unexpectedly long-running query that hogs resources? How can you tell that the query hogs resources? What if this query is business-critical and must be completed?

Increasingly, we are seeing trends towards "operational data stores" or "operational BI", where mixed workloads run on the same database. The parallel execution of workloads with different requirements on the same database poses new challenges and requires an integrated approach for workload management. As a preparatory work, this paper focuses on workload management techniques for separate execution of the different workload types. Workload management for operational data stores is ongoing work in our research collaboration.

The rest of this paper is organized as follows: Section 2 characterizes the workloads we focus on in this paper. Section 3 defines service level objectives for different workload classes. Section 4 describes how workload management is used to meet service level objectives. Section 5 summarizes prior art in industry and academia. Our proposed solution for managing OLTP workloads is presented in Section 6. Section 7 describes challenges in workload management for BI workloads. Finally, Section 8 summarizes the contents of the paper.

2 Characterization of Workloads

Table 1 characterizes business intelligence (BI) and OLTP workloads. The vertical axis describes how queries are generated. The structure of *canned* queries is fixed; the only variety stems from the parameterization of constants. This kind of query is typically issued by software clients, often by using prepared statements. In contrast, the structure of *ad hoc* queries is not known in advance, and may vary widely. Ad hoc queries may cause performance issues because they are often only executed once and thus may not be as thoroughly optimized as canned queries.

	interactive	**batch**
canned	OLTP/BI	BI
ad hoc	BI	BI

Table 1: Characterization of workloads

The horizontal axis indicates whether queries are issued *interactively* or as part of *batch* jobs. In the case of interactive invocation, the user is waiting for the results of each query before submitting the next one. A subsequent query may depend on the result of its predecessor, so even long-term monitoring of query patterns will not necessarily yield a good prediction model for query sequences. The opposite is true for batch workloads, where all queries are known in advance.

OLTP workloads are interactive, canned workloads that typically consist of a large number of small uniformly-sized queries. Results must be returned quickly for the sake of a good user experience. BI workloads, on the other hand, may contain queries from all quadrants of Table 1: interactive OLTP-like queries may be interleaved with long-running canned batch workloads that create business reports or statistics. BI workloads additionally include ad hoc queries, e. g., if a business analyst interactively performs drill-down analysis or requests a custom report to be run as an overnight batch job.

3 Service Level Objectives

From a user's point of view, a database system performs well if the performance requirements the user cares about are met. A first prerequisite is to translate user-defined performance requirements into a common set of metrics that can be obtained through monitoring. Examples of such metrics include *execution time* (the elapsed wall clock time between the start and completion of a query), *throughput* (number of successfully executed queries in a given time span), and *CPU time* (time the CPU is available for a specific query).

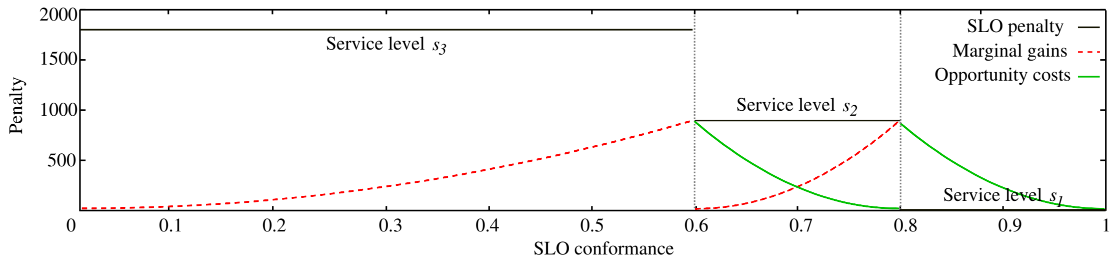


Figure 1: Visualization of SLO constraint d

A service level objective (SLO) is formed by a combination of one or more performance goals and an associated priority, which typically depends on the penalties incurred if the goals are not met. Often, these objectives do not apply to all queries, but instead must be satisfied for a certain percentage. Such *SLO conformance* metrics are defined as ratio of the number of queries that meet an SLO goal to the total number of queries.

The SLOs for the workload categories described in Section 2 differ in the way performance is measured - either in terms of individual queries or in terms of groups of queries. Users are explicitly aware of the individual response times of interactively submitted queries, but the response time for batch jobs is measured for a set of queries as a whole. Similarly, performance for canned queries (e. g., a canned report or OLTP query) tends to be measured and reported in terms of the query class as a whole, whereas ad hoc queries are measured individually.

An example for an SLO in the OLTP context is the so-called *step-wise SLO* that consists of one or more *percentile constraints*. Since users typically expect fast responses for OLTP queries, percentile constraints require $n\%$ of all service requests to be processed within x seconds. Otherwise, a penalty p for every m percentage points under fulfillment is due. A percentile constraint implicitly defines an SLO penalty function with n steps (service levels). The penalty function for a constraint d (90% in $< 5s$; $p = \$900$ per 20 percentage points, maximum penalty $\$1800$) is shown in Figure 1 (black solid lines).

In contrast, batch workloads are typically deadline-driven and may incur penalties if work is not completed before a given deadline. This translates directly into an execution time constraint. Another common SLO for batch workloads specifies a lower-bound for the throughput, i. e., the batch workload does not have a fixed deadline, but should be assigned a specific portion of the available system resources.

4 Workload Management

The goal of workload management for database systems is to increase user satisfaction by meeting SLOs. Note that in general, neither the customer nor the provider benefits from over-fulfilled SLOs, because there is no advantage to providing results before a given deadline, and the execution time requirements already ensure a responsive interaction with the DBMS. Over-fulfilling an SLO will not excuse a future SLO violation and moreover could raise unreasonable performance expectations.

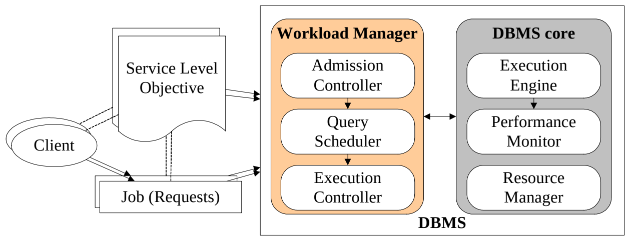


Figure 2: Generic workload management architecture

Figure 2 sketches a generic workload management architecture. The DBMS core offers the following components: the *Execution Engine*, which manages the processing of queries, the *Resource Manager*, which provides priority based allocation of resources to queries, and the *Performance Monitor* for monitoring the execution of queries. Modern DBMS offer several knobs for tuning workload performance at all points in the workload life cycle. Workload management begins with the opportunity to prevent a new query from being placed on an execution queue, continues with queuing and scheduling decisions, and includes the ability to control the execution of a running query. These mechanisms are implemented by the *Admission Controller*, the *Query Scheduler*, and the *Execution Controller*, respectively. The latter contains a rule base for identifying unexpected overload situations and deciding which workload management action should be performed for which queries. The workload management decisions are driven by the SLOs that are annotated to each batch job and interactive query of every client. The objectives must be made available for the DBMS prior to the execution of the job.

Admission control can prevent potential problem queries from being started in the first place. Query scheduling optimizes the order of execution and the number of concurrently running queries by deciding when to admit which query. Admission control and scheduling operate atop of the database layer, and can be implemented without modifying the database core engine. If the DBMS offers interfaces to control already running queries, then finer grained control of request execution is possible. The presence of execution control mechanisms that can, e. g., kill or suspend and resume queries at run-time can further improve performance. For example, killing a query that hogs system resources for an unexpectedly long time can limit the negative impact on the overall execution performance. Similarly, if the DBMS offers prioritization mechanisms for allocating resources like memory, CPU, and locks, then complex queries can be adjusted to lower priorities when necessary, leaving enough resources for newly arriving interactive queries with higher priorities.

5 Related Work

Workload Management Techniques Regarding work on workload management techniques for resource allocation, we share a focus with researchers such as [1,6,15,21], who consider how to govern resource allocation for queries with widely varying resource requirements. For example, Davison and Graefe [6] present a framework for resource allocation based on concepts from microeconomics. Their framework aims at reducing response time of queries in a multi-user environment. The central component is a resource broker that assigns operators the share of the resource they are willing to pay for. Weikum et al. [24] discuss what metrics are appropriate for identifying the root causes of performance problems in an OLTP workload (e. g., overload caused by excessive lock conflicts). They focus on tuning decisions at different stages such as system configuration, database configuration, and application tuning.

Query progress indicators (e. g., [3, 14]) attempt to estimate a running query's degree of completion. Research in this area is complementary to our goals, and potentially offers a means to identify long-running queries at early stages – before the workload has been negatively impacted.

Recently, research has been done on query suspension and resumption, e. g., [2, 4]. When a query is suspended, the DBMS releases all resources held by the query. At a later point in time, the query can be resumed, ideally without wasting a large amount of work that has been done prior to the suspension. We believe that database products will implement these techniques in the near future.

Workload Management Implementations Some workload management techniques for admission control, scheduling, and execution management policies are already implemented in products such as those by HP (e. g., Workload Manager for Neoview [8]), IBM (e. g., Query Patroller for DB2 [17], Optimization Service Center [9], zSeries [13]), Microsoft (SQL Server [16]), Oracle (Discoverer [19], Resource Manager [20]), and Teradata (Workload Manager [23]). We only present a short overview, a more detailed description can be found in [12].

Admission control uses thresholds to prevent overly-expensive queries from running in the system. Database vendors provide different metrics like optimizer costs or processing time estimates. Queries for which the values of the metrics exceed administrator-defined thresholds are either rejected or put on hold. The latter case requires the administrator to decide whether to submit the query to the database or to reject it.

Some databases allow the administrator to limit the level of concurrency on the database system and to schedule the delayed queries. There are three approaches for limiting the concurrency: limit the maximum resource utilization, restrict access to database objects like tables, indexes, and views, and limit the number of simultaneous queries in the database system. Delayed queries are typically managed in three different types of scheduling queues: FIFO, size-based, and priority-based. Size-based queues prevent short-running queries from getting stuck behind long-running queries, thus enforcing a consistent elapsed time for short queries in the presence of long-running queries. Priority-based queues enforce the preferred execution of high-priority queries compared to their lower-priority counterparts. It is the task of the database administrator to define priorities for the users in order to balance the performance of the system. Queries are then ordered in the queue according to the priority of the submitting user.

Execution control is usually implemented by using rules where a condition triggers an execution management action. The different vendors support a variety of metrics to be used in the conditions, e. g., cardinality, CPU time, number of I/Os, and elapsed wall clock time. Almost all database vendors implement some notification mechanism to inform the administrator about exceptional situations, e. g., when the elapsed time of a query exceeds a specified threshold. It is then the task of the administrator to analyze and tackle workload management problems. If the administrator does not take action, the query runs to completion. In addition to that, HP's and Teradata's Workload Manager can be configured to automatically kill a query.

6 Workload Management for OLTP Workloads

This section focuses on OLTP-style workloads that consist of a multitude of a priori unknown short-running transactions that may be started by clients at any time. Each transaction consists of a set of interactively submitted queries for which the user expects short response times. Therefore, users often negotiate SLOs similar to the step-wise SLOs introduced in Section 3, which limit the response time for a percentage of transactions. From the workload management perspective, the main challenge is to apply a query scheduling policy that meets the SLO requirements for as many users as possible. A common approach in such settings is to provide priority-based queues to manage pending queries. Typically, the priority of queries is defined externally, e. g., by a database administrator and usually depends on the priority of the respective user. If only a percentage of all queries must meet the performance requirements, static prioritization tends to over-fulfill SLOs of high-priority users because their queries are almost always processed in time at the expense of their lower-priority counterparts.

Therefore, our approach derives an adaptive penalty based on an economic model and annotates the queries with the penalty information. These penalties are used to optimize the execution order of the pending queries. We define the penalty of a query as the maximum of two economic cost functions. *Opportunity costs* (monotonically decreasing parts of the parabolas in Figure 1) model the danger of falling to the next lower service level. If the current SLO conformance converges to the next lower service level, the penalty for processing the service too late increases because delaying an additional query increases the likelihood of an ultimate SLO violation. *Marginal gains* (monotonically increasing parts of the parabolas in Figure 1) model the chance that a service class re-achieves a higher service level. If this appears to be "within reach", individual queries are increasingly penalized until eventually the higher level is reached.

The computation of penalties, scheduling algorithms for pending queries, and a performance analysis are described in [7,10]. The experimental results show the effectiveness of our novel adaptive penalization approach, which provides a higher overall SLO conformance by reducing the over-fulfillment for high-priority clients.

7 Workload Management for BI Workloads

BI workloads contain a wide variety of requests, from batch jobs to short-running interactive queries to ad hoc queries of varying complexity. For the batch jobs we focus here on deadline-driven SLOs. The optimization goal for these jobs is to minimize the time needed to complete the workload. Interactively submitted queries, on the other hand, require short response times, because users easily become dissatisfied if they must wait for responses too long. SLOs for interactive queries therefore require an execution time below a given threshold. If a batch and an interactive job are running simultaneously on the database, the challenge is to optimize the execution of all jobs subject to their SLOs. Additionally, a workload management system must be capable of dealing with ad hoc designed queries that may have unknown execution characteristics or may cause performance problems.

Admission Control One approach for optimizing the performance of BI workloads is to reject overly-expensive queries that may hog system resources and thus prevent concurrently running queries from making progress. The administrator may choose to run these queries in a controlled manner, e. g., in isolation, to minimize the impact of these problem queries on others. The challenges to be addressed are twofold: First, a threshold for identifying overly-expensive queries must be found. If the value is set too low, many queries will be rejected. A threshold that is too high may admit too many expensive queries, resulting in exceptional situations at run-time. Second, admission control requires accurate optimizer estimates and knowledge about how a query impacts concurrent queries. Query optimizers do a good job estimating the costs of queries when requirements like uniformity of data, independence of attributes, and up-to-date statistics are met. However, in the BI context, data may be heavily skewed and statistics cannot be kept up-to-date for update-intensive workloads. Therefore, the optimizer might return estimates that are off by orders of magnitude from the actual processing costs, making the estimates unusable for admission control.

Query Scheduling The task of the query scheduling component is to decide when to admit which query. Scheduling must obey inter-query dependencies, i. e., some requests cannot be reordered arbitrarily because they depend on the results of other requests. Another challenge is to determine the optimal number of concurrently running queries in the database that would maximize the usage of all system resources. The optimal number of queries in the system depends on parameters like the database configuration, the underlying hardware, and the requests themselves, and might even vary during the execution of the workload. In practice, finding an optimal solution to the scheduling problem itself is not feasible because of the high complexity and the large instance sizes containing hundreds of requests. Therefore good heuristics must be found. Additionally, a reasonable reordering of workload requests needs to consider the impact requests have on one another. Some requests are potentially working well together while others interfere with each other.

A prerequisite for scheduling is a concise representation of benefits and detriments of concurrent request execution. This information can be subsumed in a "synergy matrix" as exemplified in Figure 3. Each entry (R_i, R_j) in the two-dimensional matrix is a numerical value denoting the relative impact of the parallel execution of requests R_i and R_j. There exist several metrics for quantifying the (dis)advantages, like consumed CPU cycles, disk accesses, or execution time. In this work, we focus on the ratio of the elapsed times measured for parallel and sequential execution of requests. A value less than 1 indicates that the parallel execution is faster than the sequential execution. For example, the synergy value 0.70 for R_1 and R_2 (light gray) denotes that executing the two requests concurrently takes 70% of the time it takes to execute them sequentially. In contrast, the parallel execution of R_3 and R_4 takes longer than running them in sequence (dark gray). An empty entry ("—") in the matrix indicates that a synergy value is not yet available.

	R₁	**R₂**	**R₃**
R₂	0.70		
R₃	-	0.85	
R₄	-	0.92	1.10

Figure 3: Synergy matrix

In order to increase the quality of the scheduling results, the matrix must be populated with as many values as possible. This can be accomplished through either analysis or monitoring. The former approach applies a white box technique and is based on an analysis of the workload's requests in order to determine potential synergies before the actual execution. Sources of such synergies stem from caching behavior and multi-query optimization (MQO). In the context of MQO, extensive research has been done on identification of common sub-expressions [5, 22] and cooperative scans [25]. Drawbacks of the white box approach are that some of the required information may not be available prior to the execution of a request and that predictions errors in the analysis phase may result in incorrect assumptions about individual requests and, thus, potential synergies.

The monitoring approach treats jobs as black boxes, i. e., makes no assumptions about their characteristics. It monitors request execution and iteratively derives information about potential synergies. For example, O'Gorman et al. [18] use this approach by running all pairs of TPC-H requests both concurrently and sequentially and then comparing the number of disk accesses. A substantial drawback of the black box approach is that the synergy matrix is only populated during workload execution. Another difficulty is to infer (dis)advantages for pairs of queries if monitored data is only available for a set of concurrently executing queries.

Analysis and monitoring are complementary approaches and can be combined for better results. Prior to the execution of the batch, analysis can be used to provide an initial population of the matrix, while monitoring during run-time can be used to refine inaccurate results from the analysis and provide values for requests that cannot be analyzed or that interfere with each other unexpectedly.

Execution Control There are two major challenges for run-time execution control of queries. First, execution control needs to detect exceptional situations based on the metrics that can be monitored at run-time. Identifying a problem could be as easy as comparing the actual elapsed time of a query to a threshold provided by, e. g., the user or the administrator. More sophisticated conditions for triggering an execution control action could include additional metrics like CPU time and number of disk I/Os. Although a greater number of metrics provides a higher flexibility, monitoring the metrics may cause overhead at run-time, thus slowing down the processing of the requests. Even if a set of metrics has been identified, the challenge is to set thresholds. Practitioners with experience in workload management can attest not only the importance of good thresholds but also the difficulty of finding these values. Second, the execution management needs to choose from a set of corrective actions like killing or suspending a query. If a query is killed, the execution control needs to decide when, if at all, to resubmit it. Similarly, an appropriate policy for resuming a suspended query must be found. Of course, the execution control must obey the service level objectives, e. g., high-priority queries with tight deadlines might not be killed, even if they hog the system resources for a long time.

Experimental Framework for Workload Management To provide a more application-oriented approach for workload management, we have developed an experimental framework, introduced in [11] for evaluating the effectiveness of workload management techniques. The architecture of the framework is illustrated in Figure 2. Admission Controller, Query Scheduler, and Execution Controller represent knobs that can be adjusted to select from a variety of workload management policies and algorithms. Our framework is not limited to workload management policies already implemented by existing database systems and tools, but allows us to experiment with new workload management concepts. Furthermore, we implemented a simulator for the execution engine, which mimics the execution of a workload in a database system. We model a workload as one or more jobs. Each job consists of an ordered set of typed queries and is associated with a performance objective. Each query type maps to a tree of operators, and each operator maps in turn to its resource costs. Our current implementation associates the cost of each operator with the dominant resource associated with that particular operator type (e. g., disk or memory). We describe the life cycle of the data that drives the experimental runs in [11].

8 Summary

In this paper, we have characterized OLTP and BI workloads and identified factors in workload generation and submission that impact their service level objectives (SLOs). We outlined SLOs in the database context and the current state of the art in workload management techniques for enforcing these objectives. We summarized our contributions for managing OLTP workloads by adaptively penalizing individual queries. We looked at BI workload management where we sketched at which points in a workload's life cycle management is applicable, and presented a synergy matrix that characterizes the impact of running particular batch queries concurrently. Finally, we overviewed our experimental framework for testing the impact of the various workload management techniques on the execution of workloads. For more details about this work, we refer readers to [7, 10–12].

References

[1] M. J. Carey, M. Livny, and H. Lu. Dynamic Task Allocation In A Distributed Database System. In *Proc. of the* 5^{th} *Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 282–291, 1985.

[2] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query Suspend And Resume. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2007.

[3] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators For SQL Queries? In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 575–586, 2005.

[4] S. Chaudhuri, R. Kaushik, R. Ramamurthy, and A. Pol. Stop-and-Restart Style Execution for Long Running Decision Support Queries. In *Proc. of the 33*rd *Intl. Conf. on Very Large Data Bases (VLDB)*, 2007.

[5] S. R. Choenni, M. L. Kersten, and J. F. P. van den Akker. A Framework for Multi-query Optimization. In *Proc. of the Intl. Conf. on Management of Data (COMAD)*, 1997.

[6] D. L. Davison and G. Graefe. Dynamic Resource Brokering for Multi-User Query Execution. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 281–292, 1995.

[7] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive Quality of Service Management for Enterprise Services. *Accepted for publication in ACM Transactions on the Web (TWEB)*, 1, 2008.

[8] HP NeoView Workload Management Services Guide, August 2007.

[9] IBM Optimization Service Center for DB2 for z/OS. `http://www-306.ibm.com/software/data/db2/zos/downloads/osc.html`.

[10] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Quality of Service Enabled Database Applications. In *Proc. of the 4*th *Intl. Conf. on Service-Oriented Computing (ICSOC)*, pages 215–226, 2006.

[11] S. Krompass, H. Kuno, U. Dayal, and A. Kemper. Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls. In *Proc. of the 33*rd *Intl. Conf. on Very Large Databases (VLDB)*, pages 1105–1115, 2007.

[12] S. Krompass, H. Kuno, J. Wiener, U. Dayal, and A. Kemper. Managing Long-Running BI Queries: To Kill Or Not To Kill, That Is The Question, 2008. Submitted for publication; please contact authors for full version of paper.

[13] N. Lei. Workload Management for DB2 Data Warehouse. `http://www.redbooks.ibm.com/redpapers/pdfs/redp3927.pdf`.

[14] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL Progress Indicators. In 10^{th} *Intl. Conf. on Extending Database Technology (EDBT)*, pages 921–941, 2006.

[15] M. Mehta and D. J. DeWitt. Dynamic Memory Allocation for Multiple-Query Workload. In *Proc. of the Nineteenth Intl. Conf. on Very Large Data Bases*, 1993.

[16] Microsoft SQL Server 2005 Books Online. `http://msdn2.microsoft.com/en-us/library/ms190419.aspx`, September 2007.

[17] B. Niu, P. Martin, W. Powley, R. Horman, and P. Bird. Workload Adaptation In Autonomic DBMSs. In *CASCON '06: Proc. of the 2006 Conf. of the Center for Advanced Studies on Collaborative Research*, 2006.

[18] K. O'Gorman, D. Agrawal, and A. E. Abbadi. Multiple Query Optimization by Cache-aware Middleware Using Query Teamwork. *Softw. Pract. Exper.*, 35(4):361–391, 2005.

[19] Oracle Discoverer Administrator Administration Guide 10g (9.0.4). `http://download.oracle.com/docs/html/B10270_01/adpqta01.htm`.

[20] A. Rhee, S. Chatterjee, and T. Lahiri. The Oracle Database Resource Manager: Scheduling CPU Resources at the Application Level. `http://research.microsoft.com/~jamesrh/hpts2001/submissions/`, 2001.

[21] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. M. Nahum. Achieving Class-Based QoS for Transactional Workloads. In *Proc. of the 22*nd *Intl. Conf. on Data Engineering (ICDE)*, page 153, 2006.

[22] S. N. Subramanian and S. Venkataraman. Cost-based Optimization of Decision Support Queries Using Transient-views. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–330, 1998.

[23] Teradata. Teradata Dynamic Workload Manager User Guide, September 2006.

[24] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The COMFORT Automatic Tuning Project. *Information Systems*, 19(5):381–432, 1994.

[25] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. of the 33*rd *Intl. Conf. on Very Large Databases (VLDB)*, pages 723–734, 2007.

Towards Automatic Test Database Generation

Carsten Binnig
SAP AG

Donald Kossmann
ETH Zürich

Eric Lo
The Hong Kong Polytechnic University

Abstract

Testing is one of the most expensive and time consuming activities in the software development cycle. In order to reduce the cost and the time to market, many approaches to automate certain testing tasks have been devised. Nevertheless, a great deal of testing is still carried out manually. This paper gives an overview of different testing scenarios and shows how database techniques (e.g., declarative specifications and logical data independence) can help to optimize the generation of test databases.

1 Introduction

Everybody loves writing new code; nobody likes to test it. Unfortunately, however, testing is a crucial phase of the software life cycle. It is not unusual that testing is responsible for 50 percent of the cost of a software project. Furthermore, testing can significantly impact the time to market because the bulk of testing must be carried out after the development of the code has been completed. Even with huge efforts in testing, a report of the NIST [16] estimated the cost for the economy of the Unites States of America caused by software errors in the year 2000 to range from $\$22.2$ to $\$59.5$ billion (or about 0.6 percent of the gross domestic product).

In the early days of software engineering, most of the testing was carried out manually. One of the big trends of modern software engineering is to automate testing activities as much as possible. Obviously, machines are cheaper, faster, and less error-prone than humans. The key idea of test automation is that tests become programs. Writing such test programs is not as much fun as writing new application code, but it is more fun than manually executing the tests [3]. Automating testing is particularly attractive for the maintenance of existing software products. With every new release of a product, the implementation of a change request, or a change in the configuration of a deployment, a series of similar tests need to be carried out in order to make sure that the core functionality of the system remains intact. In fact, most software vendors carry out nightly so-called regression tests in order to track changes in the behavior of their software products on a daily basis.

In a nutshell, test automation involves writing and maintaining code and it is just as difficult as writing and maintaining application code. In fact, as will be argued, writing and maintaining test code is more difficult because it has additional dependencies. One such dependency which is of particular interest to this work is the *test database* which needs to be built and maintained together with the test code as part of a test infrastructure.

In order to deal with the complexity of managing test code, the software engineering community has developed a number of methods and tools. The main hypothesis of this paper is that testing is (to a large extent) a *database problem* and that many testing activities can be addressed best using database technology. It is argued that test code should be declarative. In particular, the specification of a test database should be declarative, rather

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

than a bunch of, say, Perl scripts. Given such a declarative specification (actually, we propose the use of SQL for this purpose), testing can be optimized in various ways; e.g., higher coverage of test cases, less storage overhead, higher priority for the execution of critical test cases, etc. Another important argument in favor of a declarative specification of test code is the maintainability and evolution of the test code. Again, the belief is that logical data independence helps with the evolution of test code and test databases in similar ways as with the evolution of application or system code. Finally, testing actually does involve the management of large amounts of data (test runs and test databases).

Of course, all software quality problems could be solved with formal methods of program verification, if they would work. Unfortunately, they do not work yet for large-scale systems and breakthroughs in this field are not foreseeable in the near future. Fortunately, test automation can benefit nicely from the results of the formal methods community, as will be shown in Section 3.

The remainder of this paper is organized as follows. Section 2 defines the many different aspects of testing. Section 3 gives an overview on how database technology can be used in order to generate test databases. Section 4 describes some research problems that we believe can be addressed by the database community.

2 The Big Picture

There has been a great deal of work in the area of software quality assurance. Obviously, one reason is the commercial importance of the topic. Another reason is that testing involves many different activities. This section gives a brief overview.

First of all, testing requires a *test infrastructure*. Such a test infrastructure is composed of four parts:

1. An installation of the *system under test (SUT)*, possibly on different hardware and software platforms (e.g., operating systems, application servers). The SUT can be a whole application with its customizations (e.g., an SAP R/3 installation at a particular SAP customer), a specific component, or a whole sub-system. One specific sub-system that we are particularly interested in is the testing of a database management system as needs to be carried out by all DBMS vendors (e.g., IBM, Microsoft, MySQL, Oracle, Sybase).

2. A series of *test runs*. A test run is a program that involves calls to the SUT with different parameter settings [13]. Depending on the kind of test (see below), the test run can specify preconditions, postconditions, and expected results for each call to the SUT. Test runs are often implemented using a scripting language (e.g., Perl), the same programming language as the SUT (e.g., ABAP, Java, or VisualBasic), or some declarative format (e.g., Canoo's WebTest [1] and HTTrace [12]). In practice, it is not unusual to have tens of thousands of test runs.

3. *Test Database:* The behavior of the SUT strongly depends on its *state*, which is ideally captured in a database. In order to test a sales function of a CRM system, for instance, the database of the CRM system must contain customers. When testing a DBMS, it does not make much sense to issue SQL queries to an empty database instance (at least not always). In complex applications, the state of the SUT might be distributed across a set of databases, queues, files in the file system (e.g., configuration files), and even encapsulated into external Web Services which are outside of the control of the test infrastructure. Obviously, for testing purposes, it is advantageous if the state is centralized as much as possible into a single test database instance and if that test database instance is as small as possible. For certain kinds of tests (e.g., scalability tests), however, it is important to have large test database instances. In order to deal with external Web Services (e.g., testing an online store which involves credit card transactions), a common technique is to make use of mock objects [4].

4. *Test Manager:* The test manager executes test runs according to a certain schedule. Again, schedules can be specified manually or computed automatically [13]. During the execution of a test run, the test manager

records differences between the actual and expected results (possibly response times for performance tests) and compiles all differences into a test report. Furthermore, the test manager controls the state of the test database if test runs have side effects.

Establishing such a test infrastructure is a significant investment. In practice, test infrastructures are often built using a mix of proprietary code from vendors of components of the SUT, from home-grown developments of test engineers, and dedicated testing tools (e.g., from Mercury or as part of IBM's Rationale Rose suite). Often, a great deal of testing needs to be carried out manually, thereby making the test team the fifth component of the test infrastructure. To the best of our knowledge, there is no silver bullet solution on how to best build a test infrastructure; likewise, there is no silver bullet solution to evolve a test infrastructure when the SUT evolves. The situation becomes even worse when considering the other dimensions of testing such as the granularity of testing (component test vs. integration test vs. system test), kinds of test (functional specification vs. non-functional requirements such as scalability, security, and concurrency), and the time at which tests are carried out (before or after deployment).

Obviously, we have no coherent solution to address all these test scenarios. Nevertheless, we believe that declarative specifications help in most cases. As an example show case, the next section shows how declarative specifications can be used in order to automate one particular activity that is important for software quality assurance: the generation of test databases.

3 Generating Test Databases

This section presents several related techniques in order to generate test databases. Both the generation of test databases in order to test application systems such as ERP and CRM systems and the generation of test databases specifically for the testing of DBMS are studied. The novel feature of these techniques is that the generation of the test databases is *query* and/or *application-aware*. This way it is possible to generate *relevant* test databases that take characteristics of the SUT and test case into account. Traditionally, generic tools to generate test databases (e.g., IBM DB2 Test Database Generator [2], [9], [8], or [14]) generate a test database based on the schema only (and possibly some constants and scaling factors). As a result, many test databases in practice are either manually constructed (possibly using the result of a generic database generator as a starting point) or constructed using scripts that must be programmed by the developer of the SUT for that particular purpose.

Query-aware and/or *Application-aware* generation of test databases has two advantages. First, the generation of test databases is simplified; only high-level declarative specifications are needed in order to generate a test database: the programming of scripts or manual adjustments are typically not needed. Second, the evolution of the system is easy. When the SUT changes and additional test data is needed, only the high-level declarative description needs to be adjusted. As shown in Section 3.2, often it is only necessary to provide an additional (SQL) query in order to specify the missing part that needs to be generated for the evolved SUT.

3.1 Reverse Query Processing

Traditional query processing takes a database and a (SQL) query as input and returns the result of that query for that specific database. The key idea of reverse query processing (RQP, for short) is to turn that process around. The input of RQP is a query, a query result and a database schema (including integrity constraints); the result is one possible database which has the property that if the query is applied to that database, the specified query result is produced. Furthermore, the generated database meets all constraints specified in the database schema.

The most obvious application of RQP is the generation of test databases. In an OLAP application, for example, RQP can be used to compute test databases from the definition of a data cube and an example report. In OLTP applications, typically, several queries are needed in order to specify a meaningful test database (Section

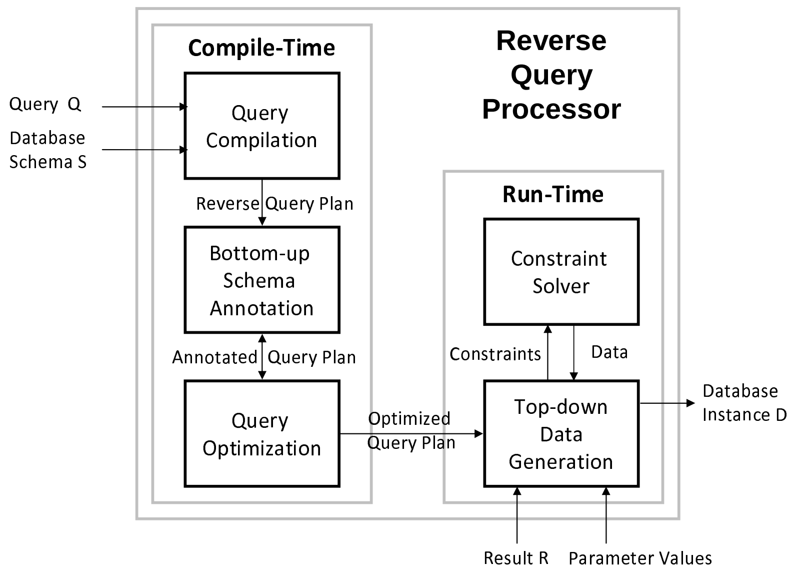


Figure 1: RQP Architecture

3.2). However, in general RQP has many more applications; e.g., security and the maintenance of materialized views.

In principle, there are many different database instances which can be generated for a given query and a result of that query. Depending on the usage of the test database, some of these instances might be better than others. For functional testing of a database application, RQP should generate a small database which satisfies the correctness criteria mentioned above, so that the running time for executing the tests is reduced. Thus, our prototype implementation tries to generate a minimal test database for a given query and a result of that query. However, other implementations are conceivable that satisfy different properties. The details of our implementation are described in [5].

Figure 1 shows the architecture of our implementation. In many ways, it resembles the architecture of a traditional (forward) query processor: A query is parsed, analyzed, optimized, and executed. Some of the key differences are that SQL queries are parsed into a *reverse relational algebra*, the optimizations are very different than in traditional query processing, and that the run time algorithms are quite different.

The reverse relational algebra can be seen as the reverse variant of the traditional relational algebra and its extensions for group-by and aggregation [10]. Consequently, executing reverse relational algebra operators at runtime involves generating data. For example, the reverse projection operator generates columns while the forward projection operator deletes columns. In order to generate data that satisfies the constraints of the query (e.g., a selection predicate) and the database schema, a decision procedure of a model checker is called by some reverse relational algebra operators. This is one example in which test automation benefits from results of the formal methods community.

In theory, reverse query processing is not decidable; that is, it is not always possible to determine whether a database exists that meets the schema and the RQP correctness condition. In practice, however, RQP is effective. For instance, RQP can be applied to all queries of the TPC-H benchmark and to all queries that we have encountered so far. For complex queries with aggregation, RQP is not trivial and involves quite complex computations. In our experiments with queries of the TPC-H benchmark, the bandwidth to generate test data on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory varied from 600GB per hour in the best cases to around 100MB per hour in the worst cases.

3.2 Multi Reverse Query Processing

In contrast to OLAP applications which implement reports that read a huge amount of correlated data from the database, OLTP applications usually implement use cases that execute a sequence of actions wherein each action reads or updates only a small set of tuples in the database. As an example, think of an online library application. One potential use case of such an application is that a user wants to borrow a book. The sequence of actions which is implemented by that use case could be as follows:

1. The user enters the ISBN of the book (where the ISBN is unique for each book of the library).
2. The system shows the details of that book.
    - Exception 1: The book is borrowed by another user. The system denies the request.
    - Exception 2: The book belongs to the closed stack of the library. The system denies the request.
3. The user enters personal data (username, password) and confirms that she wants to borrow the book.
4. The system checks the user data and updates the database.
    - Exception 3: The user has entered an incorrect username or password. The system denies the request.
    - Exception 4: There are charges on the user account that exceed a certain limit. The system denies the request.

Functional testing the implementation of such a use case means that we have to check the conformance of the implementation with the specification of the functionality [4] (i.e., the use case). Consequently, we need to create a set of test cases to test the correctness of the different execution paths of a use case. In order to execute all the test cases of an OLTP application, one or more test databases need to be created. For example, in order to test the use case above, a test database needs to be created which comprises the different types of books (i.e., books which are already borrowed by another user or not, and books which belong to the closed stack and other books which do not) and different user accounts (i.e., user accounts with and without charges which exceed a certain limit).

In order to specify a test database for the test cases of an OLTP application, one SQL `SELECT` query and one expected result are usually not sufficient. The reason is that most test cases of an OLTP application *read* or *update* different tuples in the database that are not necessarily correlated. Therefore, in order to specify the relevant values of the tuples that are read or updated by a particular test case, we suggest that a tester uses SQL as a database specification language; i.e., the tester specifies the test database for one test case by *manually* creating a set of SQL `SELECT` queries and their expected results (called test database specification). A test database which returns these expected results for all the given SQL `SELECT` queries enables the execution of a particular test case of an OLTP application. Compared to RQP where the queries are derived from the definition of a data cube, in MRQP the queries for the test database specification are not extracted directly from the code of the OLTP application. Consequently, the queries in the test database specification are independent from the SQL statements implemented by the OLTP application (i.e., the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements).

For example, in order to execute a test case for the use case discussed before where the user borrows a book successfully (i.e., no exception occurs), the test database needs to comprise a book with a particular ISBN which does not belong to the closed stack and is not borrowed by another user (i.e., the attribute $b\_closedstack$ must have the value $false$ and the value of the attribute b_uid must be *NULL*) as well as a user whose charges do not exceed a certain limit (e.g., \$20). The desirable database state, can be specified by multiple queries and the corresponding expected query results (Figure 2a) and the database schema of the application (Figure 2b). By doing so, the tester can focus on the data that is relevant (e.g., the values for b_isbn and $b\_closedstack$ specified by Q_1 and R_1) and she does not have to take care of the irrelevant data (e.g., the values for b_title).

RQP is not capable to support multiple queries and the corresponding expected results as input. Thus, in [6] we studied the problem of Multi-RQP (or MRQP for short). Unlike RQP, MRQP gets a *set of SQL `SELECT`*

```
Q1: SELECT b_closedstack, b_uid
    FROM book
    WHERE b_isbn= '12345'
R1: {<false, NULL>}


Q2: SELECT u_password, u_charge
    FROM user
    WHERE u_name='test'
R2: {<test, 0.0>}
```

(a) Test database specification

```
CREATE TABLE user (
u_id INTEGER PRIMARY KEY,
u_name VARCHAR(20) UNIQUE,
u_password VARCHAR(20),
u_charges FLOAT NOT NULL
  CHECK(u_charges>=0));


CREATE TABLE book (
b_id INTEGER PRIMARY KEY,
b_title VARCHAR (20) NOT NULL,
b_closedstack BOOLEAN NOT NULL,
b_isbn VARCHAR(20) UNIQUE,
b_uid INTEGER FOREIGN KEY
 REFERENCES user(u_id));
```

(b) Database schema S

u_id	u_name	u_password	u_charges
1	test	test	0.0

user

b_id	b_title	b_closed stack	b_isbn	b_uid
1	TitleA	false	12345	NULL

book

(c) Test database D

Figure 2: MRQP Example for OLTP testing

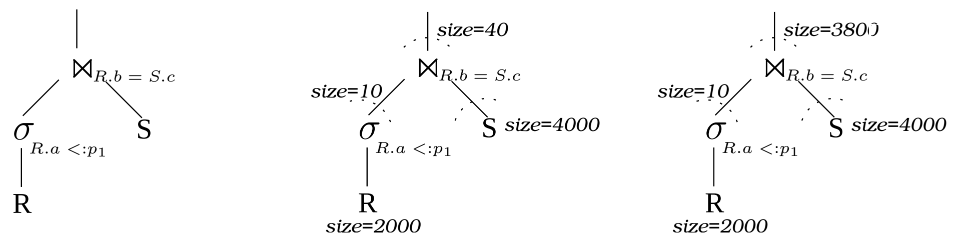
queries, the *corresponding expected query results* and a database schema as input and tries to generate one test database that returns the expected results for all the given queries. A test database which could be generated for the example above is shown in Figure 2c. In [6] we showed that MRQP is undecidable for arbitrary SQL `SELECT` queries. Consequently, we defined a database specification language called MSQL based on SQL for which the MRQP problem becomes decidable. Moreover, based on MSQL we suggested a solution for MRQP which utilizes the RQP engine discussed in Section 3.1.

3.3 Symbolic Query Processing

Symbolic query processing (SQP), which first appeared in [7], is a fusion of traditional query processing and a formal verification technique called symbolic execution [15]. In symbolic query processing, the data in a database is represented by symbols and a database query manipulates symbolic data rather than concrete data. One predominant application of SQP is to test components of DBMSs.

In the database industry, when a component or a technique is going to be integrated into a DBMS, it is necessary to validate the system correctness and evaluate the relative system improvements under a wide range of test cases and workloads. Consider that there is a new join algorithm available and a company which offers a commercial DBMS product wants to evaluate the performance of that algorithm in their DBMS product. For example, the company wants to know how much memory would be taken by such algorithm during the execution of a simple query like the one in Figure 3a. Usually, given the test query Q like the one in the figure, different *test cases* can be constructed by varying the results of the query (operators). In DBMS component testing, a test case T is a parametric query Q with a set of constraints (e.g., output cardinality) C annotated on the operators of the query. Figure 3b shows an example test case T_1 that is based on the given query Q in Figure 3a. Test case T_1 enforces that if the test query Q is executed on a test database D with two tables R and S (where R and S have 2000 and 4000 tuples respectively), then the intermediate selection $\sigma_{R.a<:p_1}$ (a is an attribute in table R and :p_1 is a parameter) and the final join result are expected to have exactly 10 and 40 tuples respectively. Test case T_1 is helpful to test how much memory the join algorithm would take when its two inputs have large size differences and the final result is small. As another example, test case T_2 in Figure 3c can test the memory consumption of the join algorithm when its two inputs have large size differences but the final result is big (3800 tuples).

Currently, testing the components of a DBMS is a manual process and thus very time consuming. It is a manual process because no tools are able to generate test databases that can fulfill the cardinality requirements of a test case. For example, in order to execute test case T_1 in Figure 3b, a tester first needs to use a normal database generator (e.g., IBM DB2 Test Database Generator, [9], [8], or [14]) to generate a test database D with



(a) The test query Q (b) Test case T_1 (c) Test case T_2

Figure 3: Examples for DBMS component testing

two tables R and S, and then *manually* adjust the content in R and S in order to ensure that the execution of Q can obtain the desired (intermediate) query results (e.g., 10 tuples should be returned by the selection $\sigma_{R.a<:p_1}$).

SQP can be used to build a database generator to automate this testing process. In fact, a test database generator called QAGen has been developed by us using SQP [7]. QAGen is a "Query-Aware" test database GENerator which generates a query-aware test database for a particular test case. It takes as input a database schema M and a test case T, and directly generates a database D and query parameter values P such that D satisfies M and $Q_P(D)$ satisfies C (where $Q_P(D)$ means the execution of query Q with parameter values P on database D, and, C are the constraints defined in T). For the example test case T_1 in Figure 3b, QAGen first instantiates the two tables R and S. In particular, table R consists of 2000 *symbolic tuples* (a symbolic tuple is a tuple containing symbols rather than concrete values; see [7] for details) and table S consists of 4000 symbolic tuples. Afterwards, the input query is evaluated by a *symbolic query engine* in QAGen. The symbolic query engine follows the paradigm of traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree [11]. In addition, the operators in the symbolic query engine manipulate input data (which are symbolic tuples) according to (1) the operator's semantics and (2) the test-case-defined constraints. On the one hand, (1) and (2) are transformed into a set of propositional constraints and the set of constraints is imposed on a subset of input tuples (and returned to the parent operator). On the other hand, the same set of constraints is directly imposed on a subset of tuples in the base tables. For the example in Figure 3b, the selection operator would impose the constraint $R.a <: p_1$ on ten of its input tuples (as well as $R.a \geq: p_1$ on all other input tuples) and return the ten tuples which pass the selection operator to the join operator. At the same time, the selection operator would impose the same constraint on the corresponding symbolic tuples in table R as well. At the end of symbolic query processing, the tuples in the base tables would capture all the requirements (constraints) defined in the input test case but without concrete data. Finally, QAGen uses a constraint solver to instantiate the constrained tuples in the base tables to obtain the final test database.

By using SQP, it could be shown that QAGen is able to generate test databases for a variety of complicated test cases. In our experiments with queries of the TPC-H benchmark, the bandwidth to generate test data on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory varied from 230MB per hour in the best cases to 6MB per hour in the worst case [7].

4 Outlook

Test automation is an important technique in order to reduce the cost and time to market of software projects. Since there are many different test scenarios with different facets, a large number of alternative tools have been developed in order to support automated testing. Most of these tools are ad-hoc and support only one particular testing activity (e.g., the generation of test reports for a large number of test runs.)

This work made the hypothesis that test automation is a *database problem*. It was shown how test databases for OLAP and OLTP applications and DBMSs can be specified using SQL queries. The ultimate goal is to support test engineers even further and to have more stable specifications of test activities.

The whole area of test automation is still in its infancy. There are still a number of open questions. We believe that in particular the following questions can be addressed using database techniques and plan to study these topics as part of future research:

- Optimize the generation of test databases; that is, generated databases with certain additional properties (e.g., the smallest possible test databases that meet the requirements or the fewest possible set of test databases).
- The evolution of test databases and test runs is a pressing issue for most software vendors who need to re-program a great deal of their test infrastructure with every major release.
- Testing distributed systems with virtualization is still a largely unexplored area; in such systems, the SUT is not known prior to deployment and changes dynamically.
- Testing the concurrency and scalability properties of a system is also largely unexplored.

References

[1] Canoo webtest. http://webtest.canoo.com.

[2] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[3] K. Beck and E. Gamma. Programmers love writing tests., 1998. http://members.pingnet.ch/gamma/junit.htm.

[4] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proc. of ICDE*, pages 506–515, 2007.

[6] C. Binnig, D. Kossmann, and E. Lo. Multi RQP Generating Test Databases for the Functional Testing of OLTP Applications. Technical report, ETH Zurich, 2008.

[7] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *Proc. of SIGMOD*, pages 341–352, 2007.

[8] N. Bruno and S. Chaudhuri. Flexible database generators. In *Proc. of VLDB*, pages 1097–1107, 2005.

[9] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Softw. Test., Verif. Reliab.*, 14(1):17–44, 2004.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[12] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Proc. of CIDR*, pages 95–106, 2005.

[13] F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal (Best of VLDB 2005)*, 16:145–164, 2007.

[14] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *Proc. of VLDB*, pages 1243–1246, 2006.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] RTI. The economic impacts of inadequate infrastructure for software testing. May 2002. www.nist.gov/director/prog-ofc/report02-3.pdf.

Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences

Leo Giakoumakis, Cesar Galindo-Legaria
Microsoft SQL Server
{leogia,cesarg}@microsoft.com

Abstract

Query optimization is an inherently complex problem, and validating the correctness and effectiveness of a query optimizer can be a task of comparable complexity. The overall process of measuring query optimization quality becomes increasingly challenging as modern query optimizers provide more advanced optimization strategies and adaptive techniques. In this paper we present a practitioner's account of query optimization testing. We discuss some of the unique issues in testing a query optimizer, and we provide a high-level overview of the testing techniques used to validate the query optimizer of Microsoft's SQL Server. We offer our experiences and discuss a few ongoing challenges, which we hope can inspire additional research in the area of query optimization and DBMS testing.

1 Introduction

Today's query optimizers provide highly sophisticated functionality that is designed to serve a large variety of workloads, data sizes and usage patterns. They are the result of many years of research and development, which has come at the cost of increased engineering complexity, specifically in validating correctness and measuring quality. There are several unique characteristics that make query optimizers exceptionally complex systems to validate, more so than most other software systems.

Query optimizers handle a practically infinite input space of declarative data queries (e.g. SQL, XQuery), logical/physical schema and data. A simple enumeration of all possible input combinations is unfeasible and it is hard to predict or extrapolate expected behavior by grouping similar elements of the input space into equivalence classes. The query optimization process itself is of high algorithmic complexity, and relies on inexact cost estimation models. Moreover, query optimizers ought to satisfy workloads and usage scenarios with a variety of different requirements and expectations, e.g. to optimize for throughout or for response time.

Over time, the number of existing customers that need to be supported increases, a fact that introduces constraints in advancing query optimization technology without disturbing existing customer expectations. While new optimizations may improve query performance by orders of magnitude for some workloads, the same optimizations may cause performance regressions (or unnecessary overhead) to other workloads. For those reasons, a large part of the validation process of the query optimizer is meant to provide an understanding of the different tradeoffs and design choices in respect to their impact across different customer scenarios. At the same time, the validation process needs to provide an assessment of regression risk for code changes that may have a large impact across a large number of workload and query types.

2 Key Challenges

The goal of query optimization is to produce efficient execution strategies for declarative queries. This involves the selection of an optimal execution plan out of a space of alternatives, while operating within a set of resource constraints. Depending on the optimization goals, the best-performing strategy could be optimized for response time, throughput, I/O, memory, or a combination of such goals. The different attributes of the query optimization process and the constraints within which it has to function make the tuning of the optimization choices and tradeoffs a challenging problem.

Large input space and multiple paths: The expressive power of query languages results in a practically infinite space of inputs to the query optimizer. For each query the query optimizer considers a large number of execution plans, which are code paths that need to be exercised and validated. The unbounded input space of possible queries along with the large number of alternative execution paths, generate a combinatorial explosion that makes exhaustive testing impossible. The selection of a representative set of test cases in order to achieve appropriate coverage of the input space can be a rather difficult task.

Optimization time: The problem of finding the optimal join order in query optimization is NP-hard [4, 8]. Thus, in many cases the query optimizer has to cut its path aggressively through the search space and settle for a plan that is hopefully near to the theoretical optimum. The infeasibility of exhaustive search introduces a tradeoff between optimization time and plan performance. The finding of the ”sweet spot” between optimization time/resources and plan performance along with the tuning of the different heuristics is a challenging engineering problem. New optimizations typically introduce new alternatives and extend the search space, often making necessary the tuning of such tradeoff decisions.

Cardinality estimation: A factor that complicates the validation of execution plan optimality is the reliance of the query optimizer on cardinality estimation. Query optimizers mainly rely on statistical information to make cardinality estimates, which is inherently inexact and it has known limitations as data and query patterns become more complex [9]. Moreover, there are query constructs and data patterns that are not covered by the mathematical model used to estimate cardinalities. In such cases, query optimizers make crude estimations or resort to simple heuristics [12]. While in the early days of SQL Server the majority of workloads consisted of prepared, single query-block statements, at this time query generator interfaces are very common, producing complex ad-hoc queries with characteristics that make cardinality estimation very challenging. Inevitably, testing the plan selection functionality of the query optimizer depends on the accuracy of the cardinality estimation. Improvements in the estimation model, such as increasing the amount of detail captured by statistics and enhancing the cardinality estimation algorithms, increase the quality of the plan selection process. However, such enhancements typically come with additional CPU cost and increased memory consumption.

Cost estimation: Cost models used by query optimizers, similarly to cardinality estimation models are also inexact and incomplete. Not all hardware characteristics, runtime conditions, and physical data layouts are modeled by the query optimizer. Although such design choices can obviously lead to reliability problems, there are often reasonable compromises chosen in order to avoid highly complex designs or to satisfy optimization time and memory constraints.

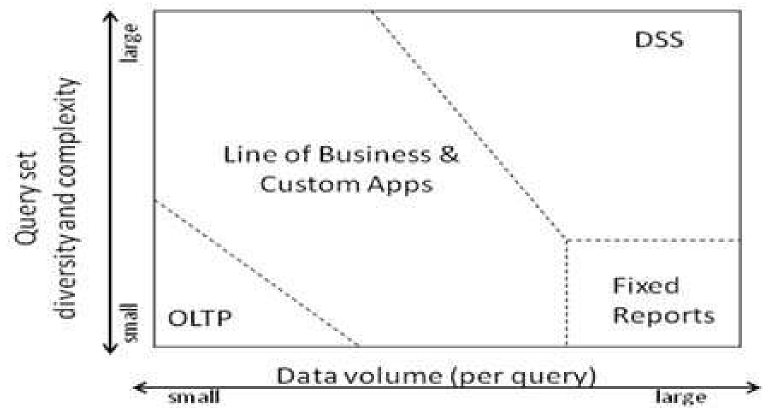

Query set diversity and complexity
large
small
DSS
Line of Business & Custom Apps
Fixed Reports
OLTP
Data volume (per query)
small
large

Figure 1: An illustration of the database application space

”Two wrongs can make a right” and Overfitting: Occasionally, the query optimizer can produce nearly-optimal plans, even in presence of large estimation errors and estimation guesses. They can be the result of ”lucky” combinations of two or more inaccuracies canceling each other. Additionally, applications may be built in a way that they rely on specific limitations of the optimizer’s model. Such *overfitting* of the application’s behavior around the limitations of the optimizer’s model can happen intentionally, when a developer has knowledge of specific system idiosyncrasies and develops their application in a way that depends on those idiosyncrasies. It can also happen unintentionally, when the developer continuously tries different ways to develop their application until the desired performance is achieved (because a specific combination of events was hit). Of course there are no guarantees that system idiosyncrasies and lucky combinations of events would remain constant between product releases or over changes during the application lifecycle. Therefore, applications (and any tests based on such applications) that rely on overfitting may experience unpredictable regressions when the conditions on which they depend change.

Adaptive optimization and self-tuning techniques: The use of self-tuning techniques to simplify the tasks of system administration and to mitigate the effect of estimation errors, themselves generate tuning and validation challenges. For example, SQL Server’s policy for automatically updating statistics [10], can be too eager for certain customer scenarios, resulting in unnecessary CPU and I/O consumption and for others it can be too lazy, resulting in inaccurate cost estimations. Advanced techniques used to mitigate the cost model inaccuracies and limitations, for example the use of execution feedback to correct cardinality estimates [14], or the implementation of corrective actions during execution time, introduce similar tradeoffs and tuning problems.

Optimization quality is a problem of statistical nature: SQL Server’s customer base includes a variety of workload types with varying performance requirements. Figure 1 illustrates the space of different workloads. Workloads on the left-bottom area of the space are typical Online Transaction Processing (OLTP) workloads, which include simple, often parameterized queries. Such workloads require short optimization times and they benefit from plan reuse. Workloads on the right side of the space may include Decision Support System (DSS) or data warehousing applications, which usually consist of complex queries over large data sets. DSS workloads have higher tolerance for longer optimization times and thus more advanced optimization techniques can be used for those. They typically contain ad-hoc queries, generated by query-generator tools/interfaces. The middle area of the application space contains a larger variety of applications that cannot be characterized as simply as the ones above. Those applications can contain a mixture of simple and more complex queries, which can be either short or long running. Changes in the optimization process affect queries from different parts of the application

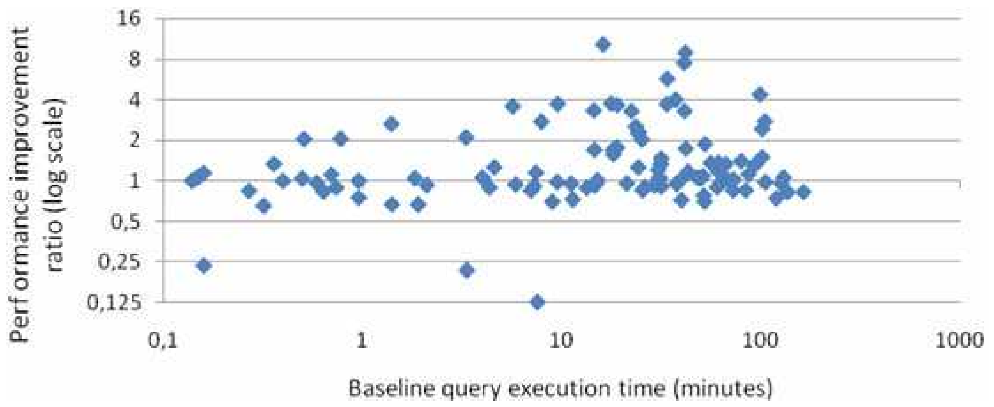


Figure 2: Performance impact of new optimizations

space in different ways, either because of shifts in existing tradeoffs and policies, or because of issues related to overfitting. Inevitably, that makes the measurement of optimization quality a problem of statistical nature. As an example, Figure 2 illustrates the results of experiments with new query optimizer features on a realistic query workload. In most cases the new features provide significant performance gains (especially for long-running queries), but they cause regressions for some parts of the workload (all points below 1 in Figure 2). Some short-running queries were affected by increases in compilation time, while a few others regressed because of a suboptimal plan choice or lack of tuning for the specific hardware used in the experiment. While in this example the benefits of the new functionality outweigh the performance regressions, there have been other cases where it was more difficult to make a judgment about the advantages vs. the disadvantages of introducing a particular new feature.

3 Query Optimization Testing Techniques

The practices of validating a software system can be typically divided in two categories: a) those that aim to simulate usage scenarios and verify that the end result of a system operation satisfies the customer's requirements and b) those that aim to exercise specific subcomponents and code paths to ensure that they function according to system design. Test cases typically aim to validate the correctness of query results, measure query and optimization performance, or verify that specific optimization functionality works as expected. We provide some examples of testing techniques from these two categories, used to validate SQL Server's query optimizer.

Correctness testing: The query optimization process should produce execution plans which are ”correct”, i.e. plans that will produce correct results when executed. *Correctness* can be validated up to some extent logically, by verifying that the various query tree transformations result in semantically correct alternatives. Additionally, it can be validated by executing various alternative execution plans using plan enumeration techniques [15] and then comparing their results with each other and/or with a reference implementation (that is typically a previous product release or a different database product). Another common practice is to run *playbacks*. Playbacks are SQL traces [1] collected from customers also used to verify correctness against a reference implementation.

Large-scale stochastic testing: Typical steps in test engineering are to identify the space of different inputs to the system under test, to recognize the equivalence classes within the input space, and then to define test scenarios that exercise the system using instances selected from these equivalence classes. As mentioned earlier, the input space for a query optimizer is multidimensional and very large. Different server configurations and query execution settings introduce additional dimensions to the input space. An effective testing technique for tackling large input spaces is to use test/query generators that can generate massive sets of test cases. The generation process can be random or can be guided towards covering specific areas of the input space or certain areas of the product. Such techniques have been very effective in testing SQL Server [7, 13, 15].

Performance baselines: The task of validating changes in the query optimizer's plan choice logic in presence of the various engineering tradeoffs can become rather difficult. A typical approach is to evaluate changes by measuring query performance against a known baseline. Industry-standard benchmarks, like TPCH [3] cover only a small part of SQL Server's functionality and contain very well-behaved data distributions. Therefore, our testing process includes a wider set of benchmarks that cover a larger variety of scenarios and product features. Normally, those benchmarks consist of test cases based on real customer scenarios. They are used for performance comparisons with a previous product release or with an alternative implementation.

Optimization quality scorecards: Although optimization time and query performance are good measures of plan choice effectiveness, they are not sufficient for an in-depth understanding of the impact of changes to the optimization process. Improvements in the optimizer's model will not always result in improvements in plan choice (for the queries included in a benchmark), but this should not necessarily mean that they have no value overall. On the other hand, new exploration rules may expand the search space with valuable new alternatives but at the cost of increased memory consumption, which may cause performance bottlenecks on a loaded server. In order to gain as much insight as possible into the impact of changes, our process includes a variety of metrics in addition to query and optimization performance. Examples of such metrics are: the amount of optimization memory, cardinality estimation errors, execution plan size, search space size, and others. These metrics can be collected across the whole set of queries included in our various benchmarks, across an individual benchmark and across segments of the application taxonomy, providing a number of different *optimization quality scorecards*.

4 Experiences and Lessons Learned

The testing techniques mentioned in this article target different classes of defects. We briefly discuss a few representative classes here, and how they correspond to the various testing techniques. We then continue with a summary of some of the lessons learned during our efforts.

Large scale stochastic testing has been effective in extending the coverage provided by regular tests. Specifically, it has contributed in eliminating *MEMO cycles* and incorrect results. MEMO cycles can occur when defects in the implementation of a set of transformation rules allow cycles to be generated in the recursive group structures. We refer the reader to [15] for an explanation of SQL Server's MEMO structure. SQL Server's code contains self-verification mechanisms to detect cycles and other inconsistencies in the MEMO structure. Therefore, the discovery of such a defect is an exercise of generating the appropriate test case. Query generators can be driven towards exploring the space of queries and query plans much further than what can be achieved by other types of testing. The combination of stochastic testing with self-checking mechanisms in the code has been very effective in detecting irregularities in internal data structures that would result to incorrect query results.

In past releases of SQL Server, the performance tuning of the database engine was done towards the final phases of product development and hence regressions in optimization time were detected late. The establishment and regular monitoring of the query optimization scorecard during the development cycle has allowed us to be proactive in identifying regressions as compared to the past. Early detection allows more time to tune the optimization heuristics towards an appropriate balance between plan efficiency and optimization time.

The combination of stochastic testing techniques and benchmarks based on realistic customer workloads has been very helpful for the development of some features of SQL Server. A case in point is the USE PLAN query hint [2], which allows forcing the optimizer to use a particular query plan that is provided by the user. While the initial prototyping and testing using real customer queries didn't indicate major issues, testing with complex queries generated by query generators showed that our technique required a lot more memory than what was anticipated. That discovery led to a number of generic improvements to the original algorithm.

The importance of a reliable benchmark: Given the statistical nature of optimization quality, it is essential that the benchmark used for making quality measurements is reliable and balanced. During the SQL Server 2000 release, our testing practice was to add a new test case every time each of our customers and partners would experience a performance regression. Adding regression tests in order to prevent future reoccurrences of code defects is a standard practice in test engineering. After following this practice for some time, the net of regression tests becomes increasingly denser and eventually provides complete coverage of areas that may have been missed in the original test plan. The regular application of the above process introduced a large number of regressions tests in our benchmark. A significant number of the regression tests corresponded to queries with large estimation errors, and included areas of query optimization with known limitations, i.e. areas where the cost model was inaccurate. During the development cycle, there were times when our benchmark was heavily affected by the performance of those tests. In some cases, legitimate improvements in the cost model would cause performance regressions. The performance of those regression tests was often unpredictable, and it could drop enough to overshadow the performance gains in other tests. At that point, it became evident to us that the practice of continuously extending our benchmark with various regression tests was problematic. While the regression tests represented areas in which customers had reported problems, they led to a benchmark that could produce inconclusive results and skew the coverage of scenarios in ways that were not well-understood. Today, we try to develop benchmarks that are more complete and balanced in terms of application type but also in terms of their conformance to the optimizer's model. If there is a specific application with which we had issues in the past and we want to track its performance, we will add a subset of that application workload into the benchmark. That helps us understand the impact of a code change on multiple queries from that application. We also try to characterize each query in the benchmark and understand its degree of conformance to the optimizer's model. That is helping us determine when a regression is caused due to a defect, a shift in optimization tradeoffs or due to side-effects of overfitting.

You improve on what you measure: The blend of application scenarios and their corresponding queries included in the benchmark influences the decisions made for the different engineering tradeoffs and eventually the tuning of the query optimizer. Initially, our testing process included a larger set of OLTP scenarios and a much smaller set of DSS-like application scenarios. OLTP customer databases were more easily accessible at the time and since they are typically smaller in size it was easier to adopt them in our test labs. Consequently, increases in compilation time during the development cycle had a significant impact across a large part of the overall benchmark, while the effect of more advanced optimizations only appeared in smaller areas. The hardware configuration used for executing the benchmark can affect the making of tuning decisions in similar ways. For this reason, the different scenarios and hardware configurations need to be defined and maintained in a way that represents the product's goals as rigorously as possible.

Test each component in isolation: The use of end-to-end query performance as the sole metric of plan choice quality has often been ineffective. First, changes made to components downstream from the optimizer in the Query Execution and Storage layers could result in end-to-end performance changes. Although the source of the regression could be pinpointed to the right component by simply checking for changes in the execution plan, there were times during the development cycle when both the execution plan and the implementation of the downstream components would change at the same time. In such cases, it was difficult to determine which component contained the root cause of the regression. Also, assumptions made by the query optimizer (such as the CPU cost for a certain operator) would change causing regressions across our benchmark. This problem was mitigated by putting in place a parallel development process, which allows development in isolated code branches. Thus, changes could be tested in isolation, and if needed, component assumptions and expectations could be adjusted before the final code integration. The concept of testing in isolation extends to testing the internal subcomponents of the query optimizer as well. In addition to evaluating the optimizer using end-to-end query performance metrics, it is valuable to be able to test each layer of the cost model independently, so that the root cause of defects can be identified quickly within the faulty subcomponent. Additionally, validating the subcomponents located lower in the optimization stack in isolation (e.g. the Statistics subcomponent) guarantees that the subcomponent located higher in the stack (e.g. the Cardinality Estimation subcomponent) operates with valid inputs and assumptions when being validated itself.

Clarify the model: It is essential that the contracts between the different components and any assumptions made in the design are crisply defined in order to validate subcomponents in isolation. For example, the cardinality estimation component operates over histograms under the assumptions of independence and uniformity. Inputs that violate those assumptions will surely result in estimation errors and possibly in suboptimal plans. Creating inputs that provide the ideal conditions expected by the cardinality estimation component allows the development of highly deterministic tests, which return accurate results. While some assumptions and contracts are fundamental and well-understood, query optimization logic can be very fine-grained. Over time, the original rationale for certain parts of that logic can fade unless it is well-documented and ensured by tests.

Agree on when a regression is a defect: As discussed earlier, it is likely that legitimate code changes can result in slower execution for some queries. It is very important that the engineering team agrees on a well-defined process on how to treat such issues, both internally as well as externally when communicating with customers. Fixing regressions in ways that do not conform with the optimizer's model and assumptions, results in code health issues and architectural debt. Supporting special cases creates instant legacy on which new applications may rely on. For this reason it is very important to have a clear definition of the optimizer's model. At the same time, every decision needs to take into account the expected impact on customer experience. Customers need to be given the appropriate tools to work around plan choice issues, and guidance through tools and documentation so that they can correct and avoid bad practices.

Design for testability. During the past four to five years of product development we went back several times to add testability features into the query optimizer in order to expose internal run-time information and add control-flow mechanisms for white-box testing. Designing new features with testability in mind is a task much easier that retrofitting testability later on. This helps in clarifying the interfaces and contracts between different subcomponents and the resulting test cases ensure that they remain valid during future development.

5 Future Challenges and Conclusions

Query optimization has a very big impact on the performance of a DBMS and it continuously evolves with new, more sophisticated optimization strategies. We describe two more challenges, which we expect will play a larger role in the future.

The transformation-based optimizer architecture of Volcano [6] and Cascades [5] provides an elegant framework, which makes the addition of new optimization rules easy. While it is straightforward to test each rule in isolation using simple use cases, it is harder to test the possible combinations and interactions between rules and ensure plan correctness. Also, with every addition of a new exploration rule, the search space expands and the number of possible plan choices increases accordingly. There is a need of advanced metrics and tools that help the analysis of the impact of such changes in the plan space. As query optimizers advance, the opportunities for optimizations that provide value across most scenarios decrease, hence optimization logic becomes more granular. There has been research that indicates that query optimizers are already making very fine-grained choices [11], perhaps unnecessarily so, given the presence of cardinality estimation errors.

Although we described query optimization testing with focus on correctness and optimality, another interesting dimension of the query optimization quality is the concept of performance predictability. For a certain segment of mission-critical applications we see the need for predictable performance to be as important as the need for optimal performance. More work is needed on defining, measuring and validating predictability for different classes of applications.

Clearly, not all the challenges that we presented in this paper have been fully tackled. The validation process and testing techniques will continue to evolve along with the evolution of the optimization technology and product goals. The techniques described in this paper allow basic validation and also provide insight regarding the impact of code changes in the optimization process. As query optimizers become more sophisticated and supplemented with more self -tuning techniques, additional challenges will continue to surface.

References

[1] SQL Server 2005 Books Online, Introducing SQL Trace. http://technet.microsoft.com/en-us/library/ms191006.aspx.

[2] SQL Server 2005 Books Online, Understanding plan forcing. http://msdn2.microsoft.com/en-us/library/ms186343.aspx.

[3] Tpc benchmark h. decision support. http://www.tpc.org.

[4] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT '95: Proc. of the 5th Intl. Conf. on Database Theory*, pages 54–67, London, UK, 1995. Springer-Verlag.

[5] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[6] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE '93: Proc. of the 9th Intl. Conf. on Data Engineering*, pages 209–218, 1993.

[7] S. Herbert H. Bati, L. Giakoumakis and A. Surna. A genetic approach for random testing of database systems. In *VLDB '07: Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 1243–1251. VLDB Endowment, 2007.

[8] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[9] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD '91: Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, New York, NY, USA, 1991.

[10] L. Kollar. SQL Server 2000 Technical Articles, Books Online, statistics used by the query optimizer in microsoft SQL Server 2000. http://msdn2.microsoft.com/en-us/library/aa902688(SQL.80).aspx.

[11] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB '05: Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 1228–1239. VLDB Endowment, 2005.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, New York, NY, USA, 1979. ACM.

[13] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB '98: Proc. of the 24rd Intl. Conf. on Very Large Data Bases*, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[14] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB '01: Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001.

[15] F. Waas and C. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *SIGMOD '00: Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 499–509, New York, NY, USA, 2000. ACM.

Testing Berkeley DB

Ashok Joshi, Charles Lamb, Carol Sandstrom
Oracle Corporation
{ashok.joshi,charles.lamb,carol.sandstrom}@oracle.com

Abstract

Oracle Berkeley DB is a family of database engines that provide high performance, transactional data management on a wide variety of platforms. Berkeley DB products are available under a dual license: an open source license and a commercial license. We discuss some of the standard testing and tuning techniques used for ensuring the quality and reliability of the Berkeley DB library, emphasizing some of the interesting testing challenges arising due to multi-platform support. Since Berkeley DB is available in source code form, it can be adapted/modified by users in the field. It is necessary to test and validate the modified version of Berkeley DB before it can be deployed in production. We discuss some testing tools and techniques provided with the Berkeley DB distribution that simplify the process of user-testing and certifying Berkeley DB ports to new platforms.

1 Introduction

Database software is complex along many dimensions: large number of features and APIs, concurrent read and write activity, fault-tolerance and recovery, performance, scalability, and reliability. In a wide variety of situations, database applications manage mission critical data, and there is an implicit assumption that the underlying data management services are well tested, reliable and correct. This article discusses some of the testing and tuning methodologies and practices used to ensure high quality for Berkeley DB, a family of embeddable database engines.

Oracle Berkeley DB [1] is a family of database engines that provide robust data management services in a wide variety of usage scenarios ranging from enterprise-class applications to applications running on mobile devices such as cell phones. Berkeley DB products are distributed under a dual license - the open source GPL-like license for open source applications, and a commercial license for closed source applications.

It is important to highlight some of the differences between an open source product such as Berkeley DB, and proprietary, closed source products. Note that Berkeley DB is not an *open development project*; Berkeley DB products are developed by a dedicated group of software engineers. Berkeley DB products are distributed in source code form, complete with a comprehensive test suite. Source code distribution adds an interesting set of development and testing challenges, since the user is free to choose from a variety of compilers and development environments to build Berkeley DB and the application. Further, a small number of users can and do modify the Berkeley DB sources, primarily for porting to new platforms; the Berkeley DB distribution includes a test suite that can be run by end users in order to validate their changes.

The rest of this paper is organized as follows. We begin with a description of each of the products. This is followed by a discussion of some of the *static testing* tools that we use internally to verify the correctness of the code. This is followed by a section on testing - this includes unit testing, stress testing, performance testing and analysis as well as ad-hoc, use-case-specific testing. Next, we discuss the Berkeley DB approach to portability, platform support and testing. Portability is particularly interesting in the Berkeley DB context because we allow and encourage our users to port Berkeley DB to the platform of their choice. The Berkeley DB distribution includes a platform test suite designed to exercise the platform-specific aspects of the port.

Oracle Berkeley DB has benefited tremendously from a large and active community of users who test the products, review the code, report problems and suggest enhancements and features. The involvement of the user community has been critical to the success of the Berkeley DB products.

2 Berkeley DB product family overview

The Berkeley DB product family consists of three products: Berkeley DB, Berkeley DB Java Edition and Berkeley DB XML. Berkeley DB products are available as libraries with simple, proprietary APIs for data access and manipulation as well as database administration. Berkeley DB does not support SQL, though it has been used as the storage engine for SQL database products. A typical Berkeley DB application makes API calls to start and end transactions, store and retrieve data as well as to perform administrative functions such as checkpoints and backups. Thus, a Berkeley DB application is completely ”self-contained” with respect to all data management activities; this enables a *zero manual administration* approach to application development. This capability is critical in a large number of applications including embedded applications, where manual administration is impossible.

Berkeley DB Java Edition is a 100% pure Java implementation whereas Berkeley DB is implemented in C; both products have similar APIs and capabilities for data management. Berkeley DB XML (implemented in C++) is an XML database engine with XQuery and XPath capabilities; Berkeley DB XML layers on top of Berkeley DB and uses it for storage, indexing, transactions and other database capabilities.

Although Berkeley DB and Berkeley DB Java Edition are very similar with respect to the features and functionality they provide, architecturally, they are quite different. From a testing point of view, porting is not as big an issue for Berkeley DB Java Edition, since the JVM is inherently portable. Both products provide indexed access to data; Berkeley DB supports B-trees as well as hash indexing whereas Berkeley DB Java Edition only supports B-tree indices. Both products support concurrent access to data. Berkeley DB permits concurrent threads, or concurrent processes or both, whereas Berkeley DB Java Edition typically supports multiple threads within a process and more limited multi-process access. Both products support transactions, including support for the various ANSI isolation levels. A row is simply an opaque *key:value* pair; Berkeley DB does not have the notion of data types, but the Direct Persistence Layer of Berkeley DB Java Edition does provide an optional schema-like capability. Interpreting the opaque contents of the row is left entirely up to the application. APIs for administrative operations like database checkpoints and backups are provided by all three products.

Architecturally, Berkeley DB is similar to the ”update-in-place” architecture of most other traditional database systems. Berkeley DB Java Edition, on the other hand, uses log-structured storage for managing on-disk data. Every change results in a new entry in the log. A separate garbage collector thread that runs in the background reclaims space occupied by obsolete data. Thus, though there are commonalities between the test suites for Berkeley DB and Berkeley DB Java Edition with respect to testing API behavior, the Berkeley DB Java Edition test suite also contains specific tests for exercising the log cleaner, ”out of disk space” scenarios, log archiving and other aspects specific to the log structured architecture of Berkeley DB Java Edition.

Berkeley DB XML, on the other hand, manages XML documents. Documents can either be stored as whole documents, or as individual nodes. Berkeley DB XML creates indices on various attributes to improve access performance. Since Berkeley DB XML is layered on the Berkeley DB database engine, it can leverage the test suite of the underlying storage engine, including the replication and high availability features. Berkeley DB XML also leverages the XML standards specifications in order to test the correctness of XML processing.

2.1 Feature sets

Early on in the history of Berkeley DB development, we made the decision to provide a variety of feature sets for the Berkeley DB products. Applications that use Berkeley DB have varying data management needs; rather than a ”one size fits all” approach, Berkeley DB products offer the user a choice of which features to use. Further, an application using the simpler feature sets can build a small footprint Berkeley DB library; this is particularly important for applications running on resource-constrained devices such as mobile devices.

The simplest feature set is called *Data Store* (available in Berkeley DB and Berkeley DB XML). This allows either a single writer or concurrent readers. The data store option is ideal for simple applications that need high performance indexed access to data without the need for read-write concurrency or transactions.

The *Concurrent Data Store* feature set allows concurrent readers and one writer, but without transactions and recovery. This is most suitable for situations where simplicity, footprint and performance of the application are more important than data consistency or integrity. In most of these situations, the data managed by Berkeley DB is either transient data, or the data can be retrieved from another (perhaps transactionally managed) data source in case of data loss.

The *Transactional Data Store* feature set provides the full set of features including concurrency, transactions, logging and recovery. This is the option of choice for applications that have stringent data consistency and integrity requirements. As mentioned earlier, Berkeley DB also provides APIs (and standalone utilities) for administrative functions such as backups, checkpoints and recovery, further simplifying the task of building zero manual administration applications.

Finally, Berkeley DB provides the *High Availability* (HA) option for applications that need multi-node scalability and extremely high availability. Berkeley DB HA supports a single read-write master and multiple reader configuration implemented via log shipping. If the master fails, a new master is elected and processing continues uninterrupted. Berkeley DB HA also supports a variety of options for improving transaction performance; for example, it is possible to commit a transaction either by writing the commit log record to the master's local disk *(commit to disk)*, or by sending reliable commit messages to a majority of the secondaries *(commit to the network)*.

The Berkeley DB design philosophy has always been to provide mechanisms, not policy. This provides tremendous flexibility to the application developer with respect to choosing the features to use as well as configuring memory, IO, network traffic, disk usage and other system resources. This level of flexibility implies multiple permutations of choices and configurations during testing and tuning.

3 Ensuring product quality

In general, we follow the *extreme programming* methodology for unit testing - implement the test first, then implement the code. This methodology results in much better code quality in the minimum amount of time. We'll discuss unit test development in more detail below.

We use a combination of code reviews and software tools in order to ensure the correctness of the new implementation. A good code review is highly effective in ensuring better code quality. Tools such as *lint* can detect potential problems such as uninitialized variables, type-incompatible assignments and incorrect arguments. Tools such as *Purify* [3] check for memory leaks and potential out-of-bounds references. Our development methodology requires that code is peer-reviewed and checked for memory leaks, adherence to language and portability standards and standard coding conventions. All *lint* inconsistencies are fixed before the code changes are approved. Since Berkeley DB products are distributed in source form, we periodically compile and build the product using a wide variety of compilers and address compiler issues and warnings.

3.1 Static testing

The term *static testing* refers to various tools for measuring code coverage, memory corruption and memory leak checkers such as *Purify*, and programs such as *lint* that are used to identify problems in the code and/or tests.

Code coverage is a very useful technique for determining the effectiveness of the test suite. In a code coverage run, the code is instrumented to monitor coverage, and then the entire test suite is run to determine which code blocks are executed, and which code blocks are not exercised. Code coverage testing is done periodically since analyzing the results and adding new tests can be a significant amount of work.

A code coverage run will highlight the lines of code that are not exercised. Rather than a ”brute force” approach to achieving very high code coverage, we focus on developing tests that target the important code paths. Code coverage results need to be interpreted carefully since they do not indicate whether every possible code *path* was exercised. For example, if there are multiple ways of reaching a particular code block, a code coverage run will identify that code block as covered even if the test only exercises one of the ways of reaching that code block. It may be necessary to use other techniques such as logging and/or using a debugger to ensure that tests exercise specific code paths. This certainly improves the quality of the tests, but may not result in increasing code coverage.

We run *Purify* periodically to identify and eliminate memory leaks. We re-compile, build and run tests periodically on many different families of systems including Linux, Windows, Solaris, HP-UX, AIX, and others. Testing starts with the compiler native to a system, but we also test non-native compilers in situations where they are commonly used, e.g. *gcc*. We also make a point of testing different versions of the same compiler. Build failures are fixed when possible and silenced when necessary to avoid the possibility of real failures vanishing in the noise of unimportant warnings. All engineers receive e-mail from automated *lint* runs daily and are responsible for fixing errors found in their code.

3.2 Regression Testing

Test execution is automated. Under normal circumstances, the test suite is run on two or three different platforms concurrently. Builds are verified at least daily. Tests generate detailed logs of the execution, and the scripts supporting the tests automatically report build and test failures by email. The QA group analyzes the results of each test run; test failures are fixed by the QA group, whereas programmatic errors are reported to the relevant developer. Code changes are logged in CVS with tracking numbers so it is usually possible for a QA engineer to pinpoint the piece of code that changed and inform the appropriate development engineer in order to address the issue expeditiously.

3.3 Unit testing

We follow the extreme programming principle of ”test first, code later” in developing unit tests. We first implement a set of tests that are designed to verify the correctness of the change. Depending on the scope of the change, developing unit tests can be a significant effort. The developer responsible for a certain feature usually develops the required unit tests, with input from other team members. The QA group expands and standardizes tests from development. Having the unit tests ready before the feature is developed has several benefits including potential improvement of the design and eliminating the possibility that the feature will go untested.

The complete unit test suite is included with each Berkeley DB distribution and can be run by the end user who needs to verify their specific Berkeley DB application implementation and deployment.

3.4 Instrumenting the code for testing

Sometimes, it is necessary to add special code to simulate certain conditions for testing (e.g. IO failure). In such cases, we instrument the code with assertions and hooks. For example, we use Java *assertions*; assertions are enabled only during debugging and testing. We also have *assert* statements that allow us to force *IOExceptions* to be thrown at specific points in the code and these points can be specified by the test program. These simulated write failures ensure that the error handling code is correct.

3.5 System and Stress testing

System and stress testing is designed to test the end-to-end behavior of the software. We use a parameterized driver program; this enables us to easily tailor a particular test run to exercise specific aspects, features and configurations. For example, it is possible to select the number of threads, the ratio of writes to reads, memory size and various configuration parameters.

The driver is normally run in randomized mode to force the testing of new combinations, and is routinely run on multi-processor machines (even slow multi-processors) to increase contention. The driver program runs for extended periods of time and exercises the various Berkeley DB APIs with the objective of finding a problem. Since the driver program is well parameterized, it is possible to use the same program not only to stress test specific aspects of the software but also to measure performance of basic operations.

Running system and stress tests is an on-going activity. Problems identified by unit tests are usually reproducible and hence relatively easy to analyze. On the other hand, diagnosing and fixing problems found by system and stress tests is harder since it is not easy to reproduce the problem. Stress tests run for extended periods of time, making it difficult to reproduce the exact state of the system at the time of the failure. Berkeley DBs extensive logging capabilities are extremely helpful in analyzing system test failures.

Testing Berkeley DB HA requires a test harness that can exercise a distributed application running on multiple nodes. We are in the process of developing a test harness based on *Erlang* [2].

Developing comprehensive system and stress tests is always a challenging task, and an on-going process. Recently, we encountered a customer issue which highlighted a limitation in one of our stress tests that exercises the *multi-version concurrency control* feature in Berkeley DB. Multi-version concurrency control requires additional memory in the buffer pool in order to store previous versions (snapshots) of database pages; when there is no more room in the buffer pool, Berkeley DB temporarily overflows the snapshots to disk. Though overflow to disk for snapshots is supported, the expectation is that the user will configure the buffer pool so that overflowing to disk is rare. In this particular customer situation, a combination of a long-running writer transaction, multiple reader transactions and a small buffer pool resulted in a large number of snapshots being written to disk. Further, there was a bug in the ”read snapshot from disk” code, which resulted in the incorrect version being returned to the transaction.

It took a significant amount of investigation to recreate the scenario and diagnose the problem, since the problem was not easy to reproduce. Fortunately, once the problem was identified, the fix was very easy (just a few lines of changed code). Needless to say, we have added stress tests to exercise the ”overflow snapshot to disk” scenario.

3.6 Performance testing and analysis

Performance testing and analysis is an on-going activity. As mentioned earlier, the system test driver program is parameterized; this enables us to use it to measure the performance of various operations.

We maintain performance data history for all releases in order to detect regressions. This is particularly helpful during the development of new features and functionality, since we can quickly identify and fix performance problems that are introduced by the new code. Our experience suggests that measuring the performance of basic operations is sufficient to identify performance regressions in most situations; a complex test is not required.

We often get requests for performance data for certain, customer-specific workloads. The customer workload usually has specific requirements for record size, number of records in the database, throughput and response time constraints and so on. Having a simple, parameterized driver program is tremendously helpful in being able to respond quickly to such requests. In most cases, it is possible to easily modify the driver program in order to approximate the specific workload and generate performance data.

3.7 Release testing

In addition to the continual testing during the development phase, we run an additional set of tests after a release is code-complete, in order to verify some of the uncommon platform configurations and to ensure that add-on modules (like *Perl* - *http://perl.com*) are tested and ready for release.

Berkeley DB has been in widespread use for more than a decade and use of historical versions is quite common. Release testing also includes upgrade tests, to verify that databases from earlier versions can be seamlessly upgraded to the new version.

3.8 Platform porting and testing

Berkeley DB products are different from most commercially available database systems because Berkeley DB is shipped in source code form (along with the source code for the tests). This makes it convenient for our users to port Berkeley DB to the platform of their choice. We often work jointly with our customers on such porting efforts.

Portability is not an issue for Berkeley DB Java Edition; the rest of this discussion applies mainly to Berkeley DB and Berkeley DB XML. By design, Berkeley DB products adhere strictly to programming language standards and have minimal dependence on platform primitives. As is the case with other portable software products, Berkeley DB isolates the operating system dependent code to a small set of code modules. This ensures that port-specific differences are localized.

Berkeley DB supports a long list of popular platforms. Each new version is released simultaneously on all supported platforms. This is achieved by continuous and frequent testing on a wide variety of platforms in a round-robin manner. Though it is not very common to find platform-specific problems, the advantage of this approach is that such problems are identified early in the development stage.

Occasionally, a customer requires Berkeley DB on a platform that is not already supported. In order to assist our users in porting Berkeley DB, we have developed a porting guide and a platform-specific test suite in C. Though not as comprehensive as the full test suite, this compact, but complete test suite is designed to thoroughly exercise the various operating system primitives that Berkeley DB uses. The tests can also be modified to suit the requirements of the underlying platform. This is especially critical when testing on resource-constrained devices such as mobile phones.

A typical customer-porting scenario is as follows. The customer will download and build the source code on the target platform. This can be an iterative edit-and-build process. After Berkeley DB is built successfully, the user can run either the full test suite (if the platform is capable) or just the platform-specific test suite. When all tests execute successfully, the user can be confident that Berkeley DB will run on the target platform.

If the user had to make changes to the code, build scripts or tests, we request them to send us the changes so that we can incorporate them into future releases.

We recently had a very positive experience where a customer worked with one of our field engineers in Japan to demonstrate that Berkeley DB could be ported to a new platform easily and painlessly. They used the porting guide and tools provided with the Berkeley DB distribution in order to compile, build and validate Berkeley DB on the new platform in less than two months. Typically, a port to a new platform of a commercial database products takes many person-months of work, so this is a remarkable achievement.

4 Tuning

The Berkeley DB philosophy is to provide mechanisms, not policy. Berkeley DB (like other DBMSs) provides a large number of ”knobs” to influence the run-time behavior and performance of the system. The user can control system parameters such as amount of memory, threads, synchronous vs. buffered IO etc. The user can also choose to enable or disable features such as transactions, locking and multi-version concurrency control.

Choosing the various parameters appropriately requires a good understanding of the system; this is further complicated because some choices have dependencies on other choices and settings.

Berkeley DB has a comprehensive statistics and logging facility that provides useful data to aid tuning. Berkeley DB documentation provides detailed information on the various parameters and settings available to the user. Further, there are several source code sample programs included with the distribution that illustrate how certain parameters may be used. The Berkeley DB discussion forums are an excellent source for getting advice and feedback on tuning Berkeley DB. In specific situations, we provide customer-specific consulting for performance analysis and tuning. Finally, having access to the Berkeley DB source code can be helpful in understanding and tuning the software. On a number of occasions, users have been able to achieve significant (ten-fold or more) improvements in performance by modifying just a few Berkeley DB parameters.

We are planning to develop a utility that will interpret the statistics and make recommendations. We are also considering integration with other comprehensive monitoring and tuning utilities such as Oracle Enterprise Manager.

5 Conclusions

Exhaustive testing is fundamental to the quality and success of the Berkeley DB family of products. We pay attention to testing, code quality and performance throughout the development cycle. In terms of lines of code, the test suite is about 40% of the lines of code in the products and it continues to evolve along with the products.

Acknowledgements

We work closely with our user community in order to improve the products as well as to port to specific platforms. Berkeley DB products have benefited tremendously from user feedback and help. A large portion of the credit goes to the excellent development team; their expertise and painstaking attention to quality, performance and overall maintainability of the code continues to be instrumental in delivering world-class products.

References

[1] Berkeley DB Documentation: `http://www.oracle.com/technology/documentation/berkeley-db/db/`

[2] Erlang: `www.erlang.org`

[3] Purify: `www.ibm.com/software/awdtools/purify`

Oracle's SQL Performance Analyzer

Khaled Yagoub, Pete Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu
Oracle USA
{khaled.yagoub, pete.belknap, benoit.dageville, karl.dias, shantanu.joshi, hailing.yu}@oracle.com

Abstract

We present the SQL Performance Analyzer, a novel approach in Oracle Database 11g to testing database changes, such as upgrades, parameter changes, schema changes, and gathering optimizer statistics. The SQL Performance Analyzer offers a comprehensive solution to enable users to forecast and analyze how a system change will impact SQL query plans and run time performance, so they can tune their system before they make the change in production. The SQL Performance Analyzer identifies potential problems that may occur and makes suggestions for avoiding any SQL performance degradation. It provides quantitative estimates of the system's performance in the new environment with high confidence and performs a comparative analysis of the response time of the SQL workload thus allowing for an easy assessment of the change. In this paper we describe the architecture of the SQL Performance Analyzer, its usage model, and its integration points with other Oracle database components to form an end-to-end change management solution.

1 Introduction

The past decade has witnessed significant advances in self-managing database technology. The major emphasis of these works [1, 5, 7] has been monitoring a currently running database system for performance regressions, diagnosing any existing performance problems, and suggesting solutions to improve such regressions. While this provides a very effective and complete solution to automatically manage database systems, there is an important aspect of query performance regressions that has been largely overlooked in the database literature: *testing the performance impact of a planned change.* In other words, how well do database systems help administrators prepare for and cope with changes?

System changes could range from simple ones like a new value for a database parameter or the addition of a new index structure to more complex changes like migrating to a newer version of the database or upgrading hardware. Since such changes are inevitable and even the smallest change to the system could have an adverse effect on the performance of certain queries, this is an extremely important problem. Since SQL performance issues are inherently unpredictable, a statement-centric solution makes sense. Users administering critical database systems need a solution to predict the negative effects of a change and take measures to avoid them. Problems left to be discovered on a live system cost the enterprise precious time and resources.

In this paper, we describe the Oracle SQL Performance Analyzer (SPA), which is our solution to the problem of controlling the impact of system changes on query performance. SPA completely automates the manual and

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

time-consuming process of testing the impact of change on potentially large SQL workloads. SPA provides a granular view of the impact of changes on SQL execution plans by executing the SQL statements in isolation before and after a change. Then it compares the SQL execution result before and after the change, and generates a report highlighting the improved and regressed SQL statements and giving precise measurements of their performance impact. Regressed statements are presented with recommendations to remedy their performance.

There have also been some efforts in the industry to address the problem of measuring performance impact caused by system changes. The Quest Plan Change Analyzer [6] relies on the Oracle explain plan command for retrieving the query plans of a set of SQL statements before and after making the desired change and then compares them. While the query plan is often a fair indicator of the actual execution cost of a SQL statement, it may not be very accurate in several situations when there is really no substitute to actually executing the SQL statement to determine its cost. Moreover, unlike SPA, the Quest Plan Change Analyzer does not consider the frequency of execution of SQL statements in a workload while computing performance impact, leading to inaccurate estimates. SPA executes each SQL before and after a change and presents SQL statements ordered by the magnitude of their change on the overall workload performance. For very large workloads, users may not have time to examine each change one by one, so separating the meaningful changes from the rest is very useful.

Hewlett Packard's LoadRunner [8] and Oracle's Database Replay [2] are two more examples of products for evaluating the impact of change on a system. However, these two differ from SPA by providing a complete system workload with timing and concurrency characteristics to a test system. In contrast, SPA computes the performance impact of a change, at the granularity of an individual SQL statement. In this context, SPA is analogous to unit-testing tools while LoadRunner and Database Replay are similar to stress-testing tools.

2 Common Usage Scenarios

SPA can be used to analyze the performance impact of a variety of system changes that can affect the performance of SQL statements. Examples of common system changes include:

- **Database upgrades including patch deployments**: Usually, database administrators (DBAs) are reluctant to upgrade to a new release of the database despite the promising new capabilities the new release offers. This is mainly because they know from past experience that any major release involves significant changes in the database's internal components, which may directly affect SQL performance.

- **Database initialization parameter changes**: The value of a specific parameter can be changed to improve performance, but it may produce unexpected results because the system constraints may change.

- **Schema changes**: Changes such as creating new indexes are intended to improve SQL performance, but they may have adverse effects on certain SQL statements.

- **Optimizer statistics refresh**: Gathering new statistics for database objects whose statistics are stale or missing can cause the optimizer to generate new execution plans. In this case, DBAs can use SPA to assess the benefit of gathering statistics.

- **Implementation of tuning recommendations**: Accepting tuning recommendations from an advisor such as Oracle's SQL Tuning Advisor [5], may require users to test the effect of the recommendations before implementing them.

- **Changes to operating systems and hardware**: Changes, such as installing a new operating system, adding more CPUs, or moving to Oracle Real Application Clusters may also have a significant effect on SQL performance.

3 SQL Performance Analyzer Architecture

Figure 1 illustrates the high level components of the SPA and their interactions with each other.

SPA takes a SQL workload as an input in the format of a SQL tuning set (see Sec. 3.1), executes every statement in the tuning set before and after making the planned change, compares the results of the two executions, and then produces a rich graphical report highlighting the impact of the change at both the SQL workload and individual SQL statement level. SPA is integrated with the optimizer's SQL Plan Management facility and the SQL Tuning Advisor (see Sec. 3.6 and 3.7) to provide support for fixing any regressions that might be caused by the change.

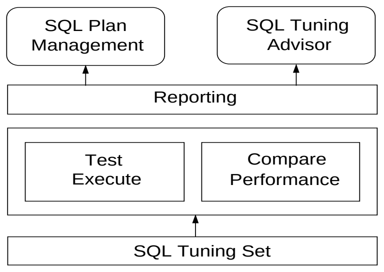


Figure 1: SPA Architecture

3.1 SQL Tuning Set

The SQL tuning set is a database object that provides a complete facility for DBAs to easily manage SQL workload information. A SQL tuning set can be used to capture and persistently store user or application-issued SQL statements along with their execution context, including the text of the SQL, parsing schema under which the SQL statement can be compiled, real bind values used to execute the SQL statement, as well as its execution plans and execution statistics, such as the number of times the SQL statement was executed.

A SQL tuning set can be populated from different SQL sources, including the cursor cache, Automatic Workload Repository (AWR) [5], existing SQL tuning sets, or custom SQL statements provided by the user. SQL tuning sets are transportable across databases and can be exported from one system to another, allowing for the transfer of SQL workloads between databases for remote performance diagnostics and tuning.

3.2 Test-execute

We believe the best way to assess the impact of a change on the performance of a SQL statement is to execute the statement before and after the change and then check if its execution time has regressed or improved. SPA test-executes SQL statements in a SQL tuning set, collects their associated execution statistics and compares them with a previous run of the same statements.

SPA employs an internal SQL service called test-execute to run SQL statements. Test-execute takes as input the text of the SQL statement to execute, actual bind values used on the production system, and a schema name to use to compile the SQL. It then performs a mock execution of the SQL statement with the goal of gathering the SQL execution plan and runtime statistics required for performance comparison. Runtime statistics include elapsed time, CPU time, I/O time, buffer gets, disk reads, disk writes, and row count. During test-execute, the SQL is executed and the produced rows are fetched until the last row in the result set, but never returned to the caller. All rows will be blocked to avoid any side effect, particularly when testing DML and DDL statements. In order to avoid updating the database state, test-execute runs only the query parts of DML and DDL statements, testing the portion of the SQL that is the most vulnerable to change.

SPA executes SQL statements once, one at a time, and in isolation from each other without regard to their initial order of execution and concurrency. This ensures that SPA performs a repeatable experiment whose results can accurately be presented on a per-SQL basis, greatly simplifying the task of interpreting the results.

Explain Plan Option: This option can be used to retrieve only the execution plans for the SQL statements before and after a change and then determine the impact of the change on the structure of the plans. This option is far cheaper than actually executing the statements.

Note that SPA still uses test-execute, but stops it after compilation of the statement to return its execution plan, which is exactly the same execution plan the optimizer would choose, had the SQL been executed with user specified bind values.

Remote Test-execute: SPA also provides the ability to perform test-execute on a remote database using database links. For example, assume that the user is upgrading from Oracle 10.2 to Oracle 11.1 and already has an 11.1 test system set up. She can use SPA on the 11.1 system to first remotely test-execute all SQL statements on the 10.2 system. Next, she can perform another test-execute, but on the local system and then compare the two sets of execution plans and runtime statistics.

To perform a remote test-execute, SPA automatically establishes a connection to the remote database using a database link specified by the user, executes the SQL statements on that database, collects the execution statistics and plan for each statement, and then stores them back in the local database for analysis and comparison.

Time Limit: To control the time spent while processing a SQL tuning set, SPA allows users to specify two time limits for test-execute: 1) A global time limit which represents the maximum duration for processing a SQL tuning set. This time limit is important, particularly, when using a large SQL workload. 2) A per-SQL time limit which is the maximum duration for the processing of a single SQL. The per-SQL time limit is used to control runaway queries. When set by the user, the same time limit applies to every SQL in the SQL tuning set.

3.3 Compare Performance

This SPA module is responsible for comparing the performance of the SQL workload before and after a change, and calculating the impact of the change on the SQL workload.

SQL Trial: The output of test-executing a SQL tuning set, i.e., the resulting execution plans and runtime statistics, are stored in the database in a container called a SQL trial. A SQL trial represents a particular experiment or scenario when testing a given change. It encapsulates the performance of a SQL workload under particular conditions of the system.

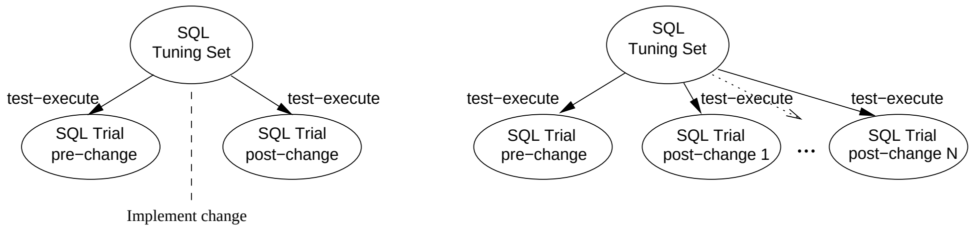


Figure 2: SPA with (a) Two SQL Trials, (b) Multiple SQL Trials

As the above diagram shows, the user can create any number of SQL trials, where each trial corresponds to the SQL workload performance data under a different change, and compare any two trials. All trials will reside in the database, thus forming a history of all testing experiments conducted by the user for a SQL workload. This is a very useful feature of SPA as it allows users to keep track of changes and perform historical performance analysis. SPA's iterative usage model is a recognition to the fact that the nature of system testing is one of one change leading to another, with each being tested in isolation until a steady state is reached.

Performance Comparison: Once performance data has been gathered under each SQL trial, the performance comparison module analyzes the differences between two trials and unmasks the SQL statements that are impacted by the tested change. The compare module measures the impact of the change on both the overall performance of the SQL workload as well as on each individual SQL statement. By default, SPA uses the elapsed time as a metric for comparison. The user can also choose from a variety of available SQL runtime statistics, including SQL CPU time, I/O time, buffer gets, disk reads, disk writes, or any combination of them as an expression (e.g., cpu_time + 10*buffer_gets). The module also compares the execution plans' structural changes of SQL between the two trials.

Change Impact Calculation: Change impact is a measure of how a system change affects the performance of a SQL statement. SPA calculates the change impact based on the difference in resource consumption across two trials of the SQL workload as follows:

$$ciw = \frac{\sum_i e_{bi} f_i - \sum_i e_{ai} f_i}{\sum_i e_{bi} f_i} \qquad cis_i = \frac{e_{bi} - e_{ai}}{e_{bi}} \qquad cisw_i = \frac{f_i(e_{bi} - e_{ai})}{\sum_i e_{bi} f_i} \tag{1}$$

ciw: change impact on the overall performance of the workload.
cis_i: change impact on individual SQL in the workload.
$cisw_i$: impact of a SQL performance change on the overall performance of the workload.
f_i: execution frequency, i.e., number of executions, of a given SQL captured in SQL tuning set.
e_{bi}: execution metric of a SQL single test-execution from the before change SQL trial.
e_{ai}: execution metric of a SQL single test-execution from the after change SQL trial.

These measurements are presented to the user through the SPA report. As a general rule, negative values indicate regressions, while positive values indicate improvements in performance.

The SQL execution frequency is used by SPA to weight the importance of each SQL statement in the workload. This allows users to correctly determine the impact on long running SQL statements that are executed only a few times as well as statements which are very fast, but repetitively executed.

3.4 Reporting

When the performance comparison and analysis are complete, all resulting data are written into the database. The end user can then review the analysis findings produced by SPA by either directly querying the exposed schema or simply requesting the analysis report from SPA.

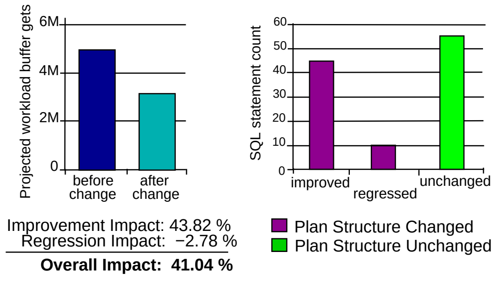


Figure 3: Example of a Partial SPA Report

The SPA report is divided into two main sections: Analysis Summary and Analysis Details. The summary section gives statistics about the overall change in performance of the SQL workload and points out the SQL statements that are impacted by the change. The detail section has an entry for every SQL statement in the SQL workload with detailed information about the SQL as well as a side-by-side comparison of the SQL runtime statistics and execution plans from the trials used in the comparison. In the report SQL statements are ordered by their change impact on the SQL workload performance.

As depicted in Figure 3, the report shows graphically the overall value of an *buffer gets* before and after making the system change, along with a second graph for the count of SQL statements whose performance improves, regresses or remains unchanged as a result of the change. Both these graphs have drill-down capabilities to view details at individual SQL statement level. The example above indicates that overall, the workload performance improved by 41.04% even though it experienced some regressions as shown by the impact of -2.78%.

3.5 SQL Plan Management

If the comparison of two SQL trials shows some SQL statements with regressed performance, SPA will recommend creation of plan baselines[1] [3] for the subset of regressed SQL using execution plans from the *first* SQL trial. This ensures that the optimizer will always use those plans for future executions of this subset of SQL statements preserving their performance, regardless of changes occurring in the system.

3.6 SQL Tuning Advisor

SPA will also recommend SQL tuning advisor [4] to fix performance problems. The SQL tuning advisor analyzes each regressed SQL statement with the goal of finding a SQL profile that will counteract the negative impact of the change. SQL profiling attempts to discover the root cause of a SQL performance problem by understanding the complex relationships in the data relevant to the execution of the SQL statement.

For statements whose performance could not be improved by the tuning advisor, the user can create plan baselines with SPA to ensure that their performance will be no worse than what it used to be before the change.

4 Usage Model

Oracle Enterprise Manager provides a graphical interface that guides a user through each of the steps mentioned in this section. We assume that a test system is available and that it resembles the production system as closely as possible. However, users can run SPA directly on the production system if, for example, they cannot afford a test system or if they have a sufficient time window to test their changes on production.[2]

4.1 Basic Testing Workflow

As Figure 4 illustrates, the testing process using SPA has the following steps:

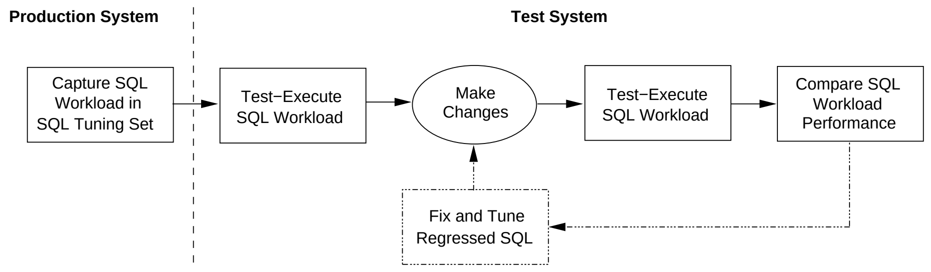

Production System
Test System
Capture SQL Workload in SQL Tuning Set
Test−Execute SQL Workload
Make Changes
Test−Execute SQL Workload
Compare SQL Workload Performance
Fix and Tune Regressed SQL

Figure 4: SPA Basic Testing Workflow

1. Capture SQL Workload: Before running SPA, users have to capture on the production system a set of SQL statements that represent the SQL workload they intend to analyze. The higher the number of SQL statements captured in the workload, the more accurate the prediction of performance changes will be. The set of SQL statements is captured and stored in a SQL tuning set. SQL tuning set provides an incremental SQL workload capture facility that enables the capture of the entire system SQL workload with minimal performance overhead. Incremental capture works by repeatedly polling the cache of currently executing SQL statements over a period of time.

[1]A plan baseline is an optimizer feature that guarantees stable performance in the face of runtime changes by maintaining a history of past execution plans for repeatable statements.

[2]Using a test system is not mandatory, but recommended since SPA test executes SQL before and after the change and this could be very resource-intensive depending on the complexity and size of the workload.

2. Transport SQL Tuning Set: After creating the SQL tuning set with the appropriate SQL workload, it is exported from the production system and imported into a test system where the system change under consideration will be tested. This can be achieved by using SQL tuning set export/import capabilities.

3. Test-execute SQL Before Change: After the SQL workload is captured and the SQL tuning set transported to the test system, SPA can be used to build the *pre-change* SQL Trial. SPA test-executes the SQL tuning set and produces execution plans and runtime statistics for each statement in the tuning set. SPA can also be run to generate SQL execution plans only, i.e., without collecting execution statistics. This technique reduces the time of SPA execution, but the results of the comparison analysis are not as complete because, without executing the SQL, it is impossible to make accurate predictions about its impact on system resource statistics.

4. Perform Change: After the pre-change trial is built, the system change to test can be implemented on the test system. This change can be any kind of change that might impact the performance of SQL statements such as a database upgrade, new index creation, initialization parameter changes, optimizer statistics refresh, etc.

5. Test-execute SQL After Change: After implementing the planned change, SPA can be invoked again to re-execute the SQL statements and produce execution plans and execution statistics for each SQL statement, a second time. This execution result represents the *post-change* trial that SPA uses to compare against the *pre-change* SQL trial. The user can also combine the explain plan option with test-execute to speed up the testing process. For example, she can start by running SPA using the explain option to retrieve the plans for all SQL in the workload and then execute only the subset of SQL whose plans changed to verify whether those plans improved or regressed.

6. Compare Performance: SPA uses the metric specified by the user and compares the performance data of SQL statements in the pre-change SQL trial to the post-change SQL trial. Finally, it produces a report identifying any changes in execution plan structures or performance of the SQL statements. The SPA analysis report explains how the tested change impacts the performance of a SQL workload and what actions can remedy the uncovered regressions.

It is important to note that neither the before nor the after SQL trial gains an undue advantage from certain system conditions such as cached data. In this case, the user can perform a dummy test execute trial to guarantee consistent caching of data across the two trials or simply use a comparison metric that is not dependent on caching such as, CPU time or buffer gets.

7. Re-iterate: If the performance comparison reveals regressed SQL statements, then the user can make further changes to fix the problematic SQL by creating SQL plan baselines or SQL profiles. The testing process can be repeated until the user has a clear understanding of the impact of the change and the corrective actions to improve the potential performance regressions. The user can then be confident to permanently make the change on production and implement the tuning actions even before the performance degradations occur.

4.2 Parameter Change Workflow

In addition to the basic testing workflow, SPA provides a predefined workflow to test database parameter alterations. This workflow enables the user to test the performance effect on a SQL tuning set when varying the value of an environment initialization parameter. Given a SQL tuning set and a comparison metric, SPA automatically creates two SQL trials and compares them. The first trial captures SQL performance with the initialization parameter set to the original value, whereas the second trial uses the new value of the parameter.

5 Conclusion

Database changes happen all the time and affect SQL performance. Therefore, one of the most important tasks for DBAs is to assess the potential impact of any changes to the database environment on SQL performance. This is a very challenging task because it is almost impossible to predict the impact of changes on SQL performance before actually implementing them in the production system. Building a thorough test bed with the ability to make reliable predictions about the impact of such changes has historically been beyond the reach of most system administrators.

In this paper, we have described SQL Performance Analyzer, which was introduced in Oracle 11g. SPA gives users the ability to measure the impact of system changes on the performance of SQL statements and fix any potential regressions before they happen in production. SPA helps DBAs build and compare different versions of SQL execution plans and runtime statistics, and then suggests tuning recommendations to overcome potential performance problems.

We have discussed the primary end user of SPA as a production DBA, but it can also be used by other types of users, such as QA testers and application developers. With SPA, DBAs have the necessary information to determine what performance changes may occur in a SQL workload and what corrective actions to undertake to fix regressions. At the same time, QA teams can use it to identify, investigate, and solve performance issues before they occur during a new application deployment. Likewise, application developers can use SPA to measure and control the risk of performance changes throughout their application's life cycle. All of these users can benefit from a comprehensive product with the ability to measure the performance impact of a change to a real SQL workload. As long as enterprises continue to expand and adapt to new environments, change will be a constant in database systems. By forecasting the impact of changes before they are implemented in production, we believe that tools like SPA eanble DBAs to clearly understand the performance ramifications of system changes and take corrective actions to avoid any potential degradations.

References

[1] S. Agrawal, N. Bruno, S. Chaudhuri, and V. Narasayya. Autoadmin: Self-tuning Database Systems Technology. *IEEE Data Eng. Bull.*, 29(3):7–15, 2006.

[2] J. Athreya and M. Minhas. Oracle Database 11g Real Application Testing Overview. Technical report, Oracle, USA, http://www.oracle.com, 2007.

[3] M. Colgan. SQL Plan Management in Oracle Database 11g. Technical report, Oracle, USA, http://www.oracle.com, 2007.

[4] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[5] B. Dageville and K. Dias. Oracle's Self-Tuning Architecture and Solutions. *IEEE Data Eng. Bull.*, 29(3):24–31, 2006.

[6] C. Fernandez and J. Leslie. Predicting and Preventing Performance Bottlenecks in Oracle 10g. Technical report, Quest Software, http://www.quest.com, 2005.

[7] S. Lightstone, G. Lohman, P. Haas, V. Markl, J. Rao, A. Storm, M. Surendra, and D. Zilio. Making DB2 Products Self-Managing: Strategies and Experiences. *IEEE Data Eng. Bull.*, 29(3):16–23, 2006.

[8] H. Packard. LoadRunner. Technical report, http://www.hp.com, 2007.

Focused Iterative Testing: A Test Automation Case Study

Mechelle Gittens, Pramod Gupta, David Godwin, Hebert Pereyra, Jeff Riihimaki
IBM Corp.

Abstract

Timing-related defects are among the most difficult types of defects to catch while testing software. They are by definition difficult to reproduce and hence they are difficult to debug. Not all components of a software system have timing-related defects. For example, either a parser can analyze an input or it cannot. However, systems that have concurrent threads such as database systems are prone to timing-related defects. As a result, software developers must tailor testing to exploit vulnerabilities that occur because of threading. This paper presents the Focused Iterative Testing (FIT) approach, which uses a repetitive and iterative approach to find timing-related defects and target product areas with multi-threaded characteristics by executing system tests with a multi-user test suite. Keywords: software testing, database management systems, multi-threaded applications

1 Introduction

IBM® DB2® for Linux®, UNIX®, and Windows® (DB2 software) is a complex distributed, multi-process, and multi-threaded system. It consisting of several million lines of source code. Execution optimization is crucial for DB2 software, and overhead from instrumentation and monitoring must be minimized.

Atomicity, Consistency, Isolation, Durability (ACID) requirements must be maintained regardless of system failures that are due to unexpected events such as power outages. After an outage, when the operating system and database restarts, the database has to replay logs of the previous database activity, so that there are no partial transactions and so that other ACID requirements are met to keep the database in a consistent state. However, in a multi-threaded, multi-process system, small timing holes[1] often exist and elusive point-in-time defects can occur. The point-in-time defects are elusive because when such an unexpected event occurs, the logs must capture concurrent events and interleave them in the manner in which they occurred so that states are repeated as they occurred previously and together.

Within this context, the DB2 software quality assurance team varied the test approaches in several ways to trigger point-in-time (timing-related) problems. These methods attempted to simulate the unexpected external issues common to databases and included: (1) Varying the processor load by running an external program to consume most of the CPU cycles available to the database server; (2) Instrumenting code to selectively slow down execution with logging overhead; (3) Changing priorities of processes; and (4) Iteratively executing commands or programs with a background workload.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]A timing hole is an unexpected point in the state space of execution for the software, where multiple threads or processes interleave in such a way as to create an incorrect logic sequence that may cause the program to hang, crash or behave incorrectly.

The main contribution of the work presented here is therefore a testing methodology with automation for complex multi-threaded database software that iteratively executes commands or programs with a background workload, since that was the most successful approach. The first three methods found defects by burdening the system resources with the various kinds of overhead mentioned in methods 1, 2 and 3. These methods however, still executed events limited sequences of events against the database, and though the additional overhead deprived the events of resources, the large set of execution possibilities (the state space) could not be representatively sampled. The iterative approach, though straightforward, randomly samples every combination of the sequences of possible events, triggering timing-related defects (TRDs) that are possible in a user environment.

The paper continues as follows. Section 2 reviews work related to testing TRDs in multi-threaded database applications. Section 3 introduces the aspects of three facilities of DB2 software that make those functions - monitoring, fast communication manager and crash recovery - suitable for testing with this method. The paper then presents the methodology for focussed iterative testing in Section 4. This section includes a description of the tool support. Section 5 summarizes the results of using the approach. Section 6 provides a summary of conclusions and potential directions for future work.

2 Related Work

Testing for database systems is normally done under the conventional and general testing approaches. These include unit, functional and system testing [6]. All of these methods have been applied successfully to database applications and there have been a plethora of discussions and work in the more general applications. However, as databases grow larger and more distributed, the context of the timing hole has become an issue unadrressed by most of the more general methods. Point-in-time interleaving of states on the stack causes transient problems. In order to encounter these problems in the various combinations and permutations of states, some testing with a statistical focus must be performed.

Random testing has proven itself cost effective and more powerful than would be thought a priori [2]. In his text also called Random Testing [3], Hamlet discusses the perception of random testing as haphazard testing, done hastily and poorly. He consequently explains that the correct meaning of random testing is one where test cases are chosen with no relationship between them. This gives statistical independence to the test points that endows statistical significance to the testing results and prediction of the expected quality of the software.

The literature covering the random testing of database systems falls into the category of randomly generating inputs for the database tests, such as the evolutionary development of queries in recent work by Bati et al. [1], where the researchers create new queries by mutating and synthesizing queries, and determining whether those queries can be used to generate further queries. Although a useful approach yielding new defects, the approach omits the issue of testing for the elusive timing hole.

Outside of the database domain other approaches to testing that handle multi-threading have existed for several years. These methods use model-checking approaches to handle the interleaving of concurrent processes and the unexpected interactions. However, the issue here is state explosion, especially with the multiplicity of states possible in the database system with tens of millions of lines of code with several interacting components. The same point-in-time timing defects are still experienced because the full population of states is still underrepresented.

Sen [5] has very recently investigated, partial-order random testing approaches that choose thread schedules at random. This approach, though yielding more defects than a nonspecific random set of tests, and demonstrating that it is useful to detect the exceptions faster than the nonspecific set of tests, was only demonstrated for three small multi-threaded programs from the Java PathFinder [8] distribution. These do not compare to complex interleaving of a large DBMS such as DB2. We cannot however say, that extrapolation and repetition of the runs would not simulate a similar execution to DB2, but since this work is recent this question will have to be explored as future work. In addition, these methods such as the one by Sen [5] did not exist when we sought to meet the TRD challenges.

In addition to the partial order methods, model-checking algorithms exist to limit the state space that must be searched to test a multi-threaded application [7]. Model checkers use the context switches that occur when a thread temporarily stops execution and a different thread starts, and systematically or iteratively suspends or binds execution of the thread at some random or arbitrary point to allow other more interesting threads to continue. One of the benefits is that the total number of executions in a program is polynomial in the number of steps taken by each thread and makes it theoretically feasible to scale systematic exploration to large programs without sacrificing the ability to go deep into the state space. This method shows potential but once again, the experimental space is preliminary with the tested programs ranging from 84 lines of code to just over 16,000 lines of code.

Having reviewed the existing work at the time and today, we found no methods to handle TRDs in a complex multi-threaded database application; however, we noted from the existing work that random testing was most suitable. We therefore created the approach presented here.

3 The Software under Study and the Components of Interest

FIT works because of the way in which functionalities such as crash recovery and monitoring are implemented in a multi-threaded system such as a DBMS. Here we explore the algorithms behind DB2 monitoring, fast communication manager, and crash recovery components and we will see why this iterative approach is particularly productive.

3.1 Monitoring

The database system monitor stores information it collects in entities called monitor elements. Each monitor element stores information regarding one specific aspect of the state of the database system. Monitor elements collect data for one or more logical data groups. A logical data group is a collection of monitor elements that gather database system monitoring information for a specific scope of database activity. Monitor elements are sorted in logical data groups based on the levels of information they provide. For this discussion, two levels are considered: database and application.

Monitor elements collect data for one or more logical data groups. These groups are collections of monitor elements that gather database system monitoring information for a specific scope of database activity. Monitor elements are sorted in logical data groups based on the levels of information they provide. The database and application levels are discussed here.

Snapshot monitor is one way that DB2 software makes element values available. Snapshots provide a point-in-time picture of the database state. During snapshot processing all relevant levels are read to complete the snapshot. As a result, for a database snapshot, the following events occur:

1. Read element values from the application-level structure. This will contain values for all terminated applications since the database activated.

2. Iterate through each application currently connected to the database. For each of these: (a) Read the element values from the application level structure. This structure contains values for all agents that have disassociated from this application. (b) Iterate through agents currently associated with the application. For each of these read the element values from the agent-level structure.

In the snapshot output, the rows_read element will contain the total from all these levels.

Event monitors are a second facility with which DB2 software makes element values available. Event monitors provide a real-time trigger-based monitoring capability. The event monitor infrastructure buffers records, using two internal buffers, before writing them to disk.

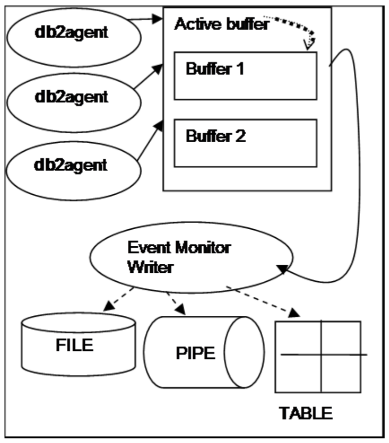


Figure 1: Generating event records for a STATEMENT event monitor

Figure 1 illustrates how event records are generated for a STATEMENT event monitor. Three applications are connected to the database, each having a single agent working on its behalf. As each application executes SQL statements, their agents generate new statement event monitor records and insert them into the active buffer, which is Buffer 1. When Buffer1 has filled up, a message is sent to the event monitor writer instructing it to process all records in Buffer 1. During this period, the “active” buffer is switched from Buffer 1 to Buffer 2 and the three applications will begin filling up Buffer 2. The Fast Communication Manager (FCM) event monitor writer, having received the message, processes Buffer 1 and inserts the data into files, if the event monitor is a FILE event monitor; or a named pipe, if the event monitor is a PIPE event monitor; or SQL tables, if the event monitor is a TABLE event monitor.

3.1.1 Opportunities for FIT

One of the challenges facing monitor processing comes from the transient nature of the memory it needs to read. Busy systems can find applications constantly starting up or terminating. This results in application memory being allocated and freed. Moreover, during the processing of a snapshot, agents may be in the process of either associating with applications or disassociating from applications. At the start of snapshot processing, an agent may be working on behalf of one application but by the end of the snapshot processing, it may be working on some other application.

To protect the ACID properties of a monitor operation and in particular the consistency and isolation of the monitor action, monitor processing must lock resources. Resources are locked before that resource can be accessed, and the lock protects the integrity of the monitor data and ensures that memory does not “disappear” while being read (resulting in crashes). Locking of a resource involves acquiring a “latch”, which is an internal mechanism for controlling concurrent events and the use of shared system resources. The protocol surrounding the locking of resources is strict, and resources must be locked and unlocked in a certain order and monitor processing must adhere to such protocols. If resource locking protocols are broken, the system can “hang”, so extra care must be taken to ensure that protocols are followed.

This requirement for ordering of events and the strict requirements of the protocols means that timing problems are more probable. It also results in a requirement to test these protocols by concurrently to increase the probability of performing locking and unlocking events out of sequence. The FIT method, which samples several event combinations and introduces resource constraints just as in a locking situation iterates through such probabilistic populations of events.

Multi-partition instances in the Data Partition Feature (DPF) environment present monitor processing with other challenges. The activating or deactivating of event monitors requires the coordination of activity across all partitions. This involves sending messages to all partitions, waiting for all the partitions to perform the activation or deactivation and respond with success or failure, and coordinating the replies. In addition, global snapshot processing requires similar coordination. Messages must be sent to all partitions, snapshots executed locally on each partition with output sent in replies to the messages, and then replies merged into a single snapshot output stream. This processing must prove resilient to partitions activating and deactivating and dropped messages. Additionally, extra care must be taken to ensure that resources are not locked on one partition while messages are sent to other partitions. Failure to ensure the coordination of the deactivation and activation, as well as locking and messaging transmission, may result in severe software defects. In a threaded environment, the iterative approach in FIT creates a sample of situations where such interleaving and order can be disturbed.

3.2 Fast Communication Manager (FCM)

In order to achieve high performance and scalability, DB2 software provides two operating modes for parallel execution of activities. (Both rely on the availability of multiple CPUs for processing.) One of these configuration types is intra-query parallelism or SMP (Symmetric Multi-Processor configuration).

SMP works by generating SQL execution plans whereby portions of an SQL statement are divided into individual sections, which can be executed concurrently and independently by multiple processes/threads. The second type of configuration is Data Partitioning Feature (DPF). This configuration allows for the partitioning of data across multiple DB2 nodes. Each DB2 node is responsible for managing one data partition. The architecture where each DB2 node and its associated data partition are independent from other partitions is commonly referred to as “shared nothing”. It permits function shipping whereby SQL and non-SQL operations are directed to those nodes where the target data is held for local operation. Where multiple data partitions are involved, parallel processing occurs (with each DB2 node only working with its subset of data). SMP and DPF can also be combined within the same instance of DB2 software. Both these configurations require fast and efficient internal communication facilities.

A user may configure a DB2 instance with multiple nodes residing on the same host machine. Such configurations are described as Multiple Logical Nodes (MLNs). In such configurations, communication between DB2 nodes residing on the same host occurs through shared memory.

3.2.1 Fast Communication Manager Design

The fast communication manager component includes FCM resources, FCM receiver and sender conduits, connection management and node failure support. **FCM resources** are allocated from a separate shared memory segment that is allocated at start-up time by DB2. The two main FCM resources are buffers – which store communication data - and channels - which are the terminal points in communications. Each node on a DPF instance will have at least one **FCM receiver conduit** for incoming messages and one **FCM sender conduit** for outgoing messages. Connections are established on demand with **connection management**. The first indication of user activity on a node drives FCM to initiate communication with every other node configured in the instance. This ensures optimal performance and security of inter-node communication. **Node failure support** involves interrupting applications with dependencies on nodes that have lost their connection to the system and cannot be contacted. The FCM node failure-recovery facility allows applications to process node failures asynchronously from each other.

3.2.2 Opportunities for FIT

There are aspects of shared memory, failure recovery, connection management and conduit management that may create opportunities for challenges due to unusual circumstances such as frequent system interruptions, frequent reconnections and unusual resource deprivation. An example of this is monitor running with FCM. With FCM, the single shared resource pool is created on each host on the MLNs to facilitate communication between logical nodes. In a multi-process environment, the interleaving of events may compete for memory. FCM, with the conduits establishing their own connections and resulting connection management, is designed to handle this interleaving adequately. In the case of node failure, and other unexpected events however, the probabilities of unexpected and sometimes incorrect interleaving may be increased. In this case, FIT can deprive resources and increase the samples of code failure events with memory management and connection management choices made by FCM. This will increase the probability of finding TRDs in FCM.

3.3 Crash Recovery

The third important feature of DB2 software that has been found suitable for testing with the FIT approach is the crash recovery feature [4].

Since units of work on a database can be interrupted unexpectedly, if an interruption occurs before all of the transactions in the unit of work are completed and committed, the database is left in an unusable state. Crash recovery moves the database back to a consistent and usable state by rolling back incomplete transactions and completing committed transactions still in memory.

Transaction failures result from conditions that cause the database or the database manager to end abnormally. Partially completed units of work that have not been flushed to disk at the time of failure, leave the database in an inconsistent state. Following a transaction failure, the database must be recovered. Conditions that may result in transaction failure include a power failure on the machine, causing the database manager and the database partitions on it to end abnormally; a hardware failure such as memory corruption, or disk, CPU, or network failure; or a serious operating system error that causes the DB2 application to end abnormally.

3.3.1 Opportunities for FIT

The conditions mentioned above that lead to transaction failure can create vulnerabilities and timing issues that should be found in testing. Order dependency is important because logs of the events that ran earlier must be replayed either to roll back partial transactions or complete uncommitted transactions in memory. The same issues arise because of parallelism and the need to replay logs in correct sequence. FIT is able to exploit the sequencing vulnerabilities by sampling from a large number of execution sequence possibilities.

In addition, transactions are logged while they occur, whether or not the transactions are committed. Transactions go from the log buffer to log files (transactional logging) before any data is written from the buffer pools to the database structures. Challenges can occur again in a multi-threaded environment because of the interleaving of events and the need to separate a given sequence when a problem occurs. This is a standard protocol, but problems can only be revealed with mass repetition of such logging of parallel processes. FIT tools facilitate execution of a large number of iterations of runtime and recovery scenarios and hence increase the probability of finding defects that occur during a particular sequence of events.

4 Methodology

The FIT approach hinges on repetition and resource deprivation, and as a result, automation is vital. The methodology presented is useful to those with large multi-process, multi-threaded software testing concerns, with vast combinations of possible executions and resource constraints are likely to trigger problems.

The approach is run as a number of controlled iterations on any machine and operating system combination. The iterations proceed with the following steps.

Step 1: Run random concurrent database test suites in the background to stress the supporting hardware, operating system and database, while varying the configuration parameters of the database, the size of the database,

the operating system, and the nature of the test suite being monitored (for example with a workload that tests monitoring functionality such as snapshot, crash recovery functions, or the fast communication manager). Since the configuration parameters control features crucial to monitoring, FCM and crash recovery (such as memory distribution (including sorting and locking), parallelism, I/O optimization (asynchronous page readers and writers), many aspects of logging (file size, buffer size), and recovery), it creates circumstances that are well-suited for uncovering potential software errors.
Step 2: Deliberately crash the database server by issuing a kill signal to the operating system.
Step 3: Restart the database server and the database
Step 4: Check for data integrity problems, database crashes (traps), and database hangs.
Step 5: If any problem is found, then exit and notify the tester via electronic mail alert. Else repeat from Step 1.

Supporting tooling was created to run several parallel processes and vary parameters. The tools execute the algorithm for the approach above and stress the monitor heavily by using the command line processor interface in one tool and the DB2 application-programming interface (API) in another tool to invoke the snapshot and the event monitor functions. The tool allows the user to control the number of iterations and is available for the UNIX and Windows platforms.

Another supporting tool runs the algorithm with the crash recovery procedure. The crash recovery tool runs the algorithm with several thousand crash recovery iterations by crashing the DB2 instance on all partitions and then restarting the database. This tool allows the tester to specify the number of crashes and is written for both the UNIX and Windows platforms.

5 Results

After applying the FIT method, defect detection improved significantly, and therefore increased tester productivity. Automation was key to the approach since the FIT tools were executed in scenarios with multiple databases with concurrent test suites running for extended periods (for example overnight).

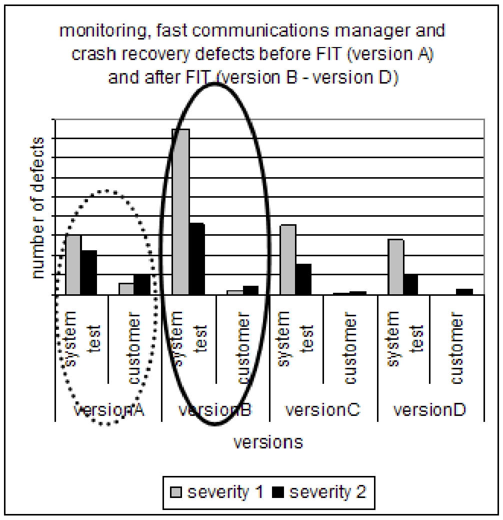


Figure 2: Defects for timing-dependent components found in system testing and by customers as an example of the defect discovery occurring between the introduction of FIT in version B and beyond

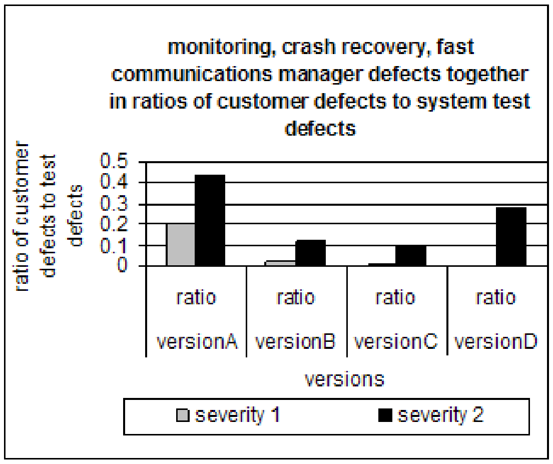


Figure 3: All defects for components most affected by timing issues as ratios between customer defects and test defects

Figure 2 shows the general trend for the three components in the DB2 product where FIT was used. The areas indicated by the solid circles (versus the dotted circles) show the increase in severity 1 and severity 2 defects found in system test versus by the customer after FIT in version B. Severity 1 defects cause the system to become unavailable, and severity 2 defects cause problems that hinder work but may be worked around. In addition, the ratio of customer defects versus system test defects decreases. That is, after FIT, testers find more defects and customers find fewer. Figure 3 shows the ratio between defects found in testing in the components before (version A) and after (version B and on) FIT; and the defects found by customers.

Since the tool repeats the same steps nondeterministically with each iteration, the tool increases the probability of hitting the same defect over time. Previously where TRDs occurred, the causes were difficult to record. This iterative method, using the same non-deterministic workload, increased the probability of hitting defects, therefore making it easier to reproduce the defects for debugging.

Additionally, the number of successful iterations before detecting a defect provided management with a quantitative and objective measure of the quality of the software. The number of iterations is independent of CPU speed. As a result, the number of successful iterations differs from the execution time for a workload, which depends on the CPU speed. This led to the formal requirement of a minimum number of successful iterations before the product could be shipped.

The approach was first applied to monitoring, and because of its success it was extended to crash recovery and fast communication manager. It is suitable to any areas of DB2 software where timing-related concerns may exist. These are areas involving communications between different nodes, data transfer between nodes and monitoring of nodes.

5.1 FIT Overhead

The mean number of iterations required to find a defect varies with time. The number is small at the beginning of the test cycle and grows towards the end. The value of the number depends on the component under test. The ideal value is infinity, where no defects are found and every iteration is successful. However, in reality, the principle of “good enough” reliability discussed in software reliability engineering is used, and test management decides on a particular target value for exiting the test cycle. The higher this target value, the higher the reliability of the component is deemed.

There are two types of overhead for this technique: (a) extra tooling effort in automating the FIT approach by making a FIT tool and (b) extra computational effort of running a FIT tool. For (a), the extra effort is mostly a one-time effort at the beginning of the test cycle. However, this tooling effort is small compared to the effort spent on the entire test cycle. For (b), the extra effort is negligible since most of the time of the FIT tool is spent running the test and only a small amount is spent preparing for each new iteration.

6 Conclusions

The FIT approach has many benefits. They included an increase in the number of timing-related defectTRDs found in system testing as opposed to such discoveries by customers and a new objective measurement for the quality of the DB2 timing-affected components that is independent of the platform on which the software runs.

One of the side effects of the quantitative measure of quality is the ability to determine the mean number of iterations to failure for components tested with the FIT approach. For crash recovery functionality, for example, this number has improved over tenfold since this method was employed and the measurement taken.

The first major lesson is that the automation created to support FIT is crucial because the large state space has meant that the ease in executing the algorithm has had many unexpected but welcome effects. For example, FIT has been able to identify stability regression defects in real time during the development cycle, that is, whilst testing for build-to-build regressions. More specifically, during continuous crash recovery testing, when testing from one build to the next, there are sometimes regressions in both runtime and crash recovery testing. These are all rooted in recently integrated code that is intended to correct a defect or add new functionality. Because of the existing automation and the ease of implementing the method, these build-to-build regressions were easily discovered.

There was also a significant return on the initial investment to create the tools, since with the tools several workloads could be run simultaneously and easily, and left to sample the execution state space for crash recovery, monitoring, and fast communication manager. The tools would easily find new defects while running over many days. Instead of requiring the previous tester time to explore the state space, the tools are left fishing for defects on their own. This is inexpensive.

"Build it and they will come" – this quote does not apply to testing tools. Complicated tools, however useful, are left to gather dust. One of the other important points for FIT beyond automation was the ease of use of its automation. If setup of tooling is complicated and running the tool is complicated, then it will likely be used sparsely in testing. The FIT tools were carefully crafted to avoid such difficulty and have been intensely employed to validate the product. This has meant that many more defects have been found. The tooling has also been built so that one tester can easily set it running on many machines. Moreover, the tool alerts the tester when a defect is found; hence the tester does not have to monitor the test systems continuously

The underpinning factor has been the feasibility of this approach to test automation. This has resulted in returns that far exceed the investment. The future work with this approach will be in extending it to additional areas of the DB2 product.

References

[1] H. Bati and L. Giakoumakis and S. Herbert and A. Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*,pages 1243-1241,Vienna Austria, September 2007. VLDB Endowment

[2] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*,SE-1-(10):438-443. July 1984.

[3] R. Hamlet. *Random Testing*. Wiley. 1994.

[4] DB2 for Linux, UNIX, and Windows . http://publib.boulder.ibm.com/infocenter/db2luw/v9. IBM Press, Current February 2008.

[5] Sen K. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international Conference on Automated Software Engineering*, pages 323-332, Atlanta, Georgia, USA, November 2007. ACM New York

[6] E. Kit. *Software Testing in the Real World*. Addison-Wesley Professional. 1995

[7] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multi-threaded programs. In *Proceedings Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 446 - 455. San Diego, California, USA, June 2007. ACM, New York, NY.

[8] W. Visser and K. Havelund and G. Brat and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*. IEEE Computer Science Press. September 2000.

Trademarks

ICDE'08

24th IEEE International Conference on Data Engineering

April 7-12, 2008, Cancún, México

CALL FOR PARTICIPATION

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments.
The **24th IEEE International Conference on Data Engineering** will provide a forum for:

- sharing research solutions to problems of today's information society
- exposing practitioners to the latest research, tools, and practices
- raising awareness in the research community of the problems and challenges of practical applications of data engineering
- promoting the exchange of data engineering technologies and experience among researchers and practitioners
- identifying new issues and directions for future research and development work
- the exchange and discussion of new ideas and for interacting/networking with peers

HIGHLIGHTS

- 3 ***Keynotes***
- 6 ***Advanced Technology Seminars***
- 75 ***Research*** papers with *full presentations*, and 44 ***Research*** papers with *short presentations*, out of over 600 submissions
- 10 ***Workshops*** in conjunction with the main conference
- 3 ***Panels***
- 75 ***Posters*** out of 650 submissions
- 13 ***Industrial*** papers, out of over 50 submissions
- 23 ***Demos,*** out of over 60 submissions

KEYNOTES

- Hector Garcia-Molina (Stanford University, USA), on ***PhotoSpread: A Spreadsheet for Managing Photos***
- Martin Kersten (CWI, The Netherlands), on ***The Database Architecture Jigsaw Puzzle***
- Luis von Ahn (Carnegie Mellon University, USA), on ***Human Computation***

WORKSHOPS

- ***Self-Managing Database Systems*** (SMDB)
- ***Mining Multimedia Streams in Large-Scale Distributed Environments*** (MMSDE)
- ***RFID Data Management*** (RFDM)
- ***Ranking in Databases*** (DBRank)
- ***Methodologies, Architectures and Systems for Information Integration*** (IIMAS)
- ***Secure Semantic Web*** (SSW)
- ***Data and Services Management in Mobile Environment***s (DS2ME)
- ***Networking Meets Databases*** (NetDB)
- ***Data Engineering for Blogs, Social Media, and Web 2.0***
- ***Similarity Search and Applications*** (SISAP)

ADVANCED TECHNOLOGY SEMINARS

- ***Mobile and Embedded Database Systems and Technology***

Anil Nori (Microsoft Corp.)

- ***Data and Metadata Alignment: Concepts and Techniques***

Lise Getoor (University of Maryland) and Renee Miller (University of Toronto)

- ***Exploring the Power of Links in Scalable Data Analysis***

Jawei Han (University of Illinois, Urbana-Champaign), Xiaoxin Yin (Google) and Philip Yu (IBM T.J. Watson)

- ***Stream Processing: Going Beyond Database Management Systems***

Sharma Chakravarthy (University of Texas, Arlington)

- ***The Java Persistence API (JPA): Technology, Standards, and Implementations***

Patrick Linskey (BEA Systems, Inc.)

- ***Performance Evaluation in Database Research: Principles and Experience***

Ioana Manolescu (INRIA Futurs) and Stefan Manegold (CWI)

Venue: CANCUN

ICDE 2008 will take place at **Cancún**, a truly unique spot nestled in the heart of the Mexican Caribbean. Cancún's Hotel Zone is a 14 mile long island shaped like a "7" and connected to the mainland by bridges at either end. Some 25% of this area's natural surroundings are protected in ecological reserves and holds the second largest reef system in the world. Part of its cultural heritage can be seen in dozens of remains of ancient Mayan cities—some more than 2,600 years old—encompassing more than 5,000 buildings or temple mounds.
The conference hotel, **Presidente InterContinental Cancún Resort**, is all you could ever want in a tropical retreat: five-star luxury, premium restaurants and the best, picture-perfect beach in the peninsula. Central shops and nightlife are 10 minutes away, Isla Mujeres is just across the water, and there's a golf course next door.

For more information, visit www.icde2008.org

IEEE COMPUTER SOCIETY · IEEE · hp invent · Microsoft · CIC · IBM · SAP · YAHOO!