Requirements-driven database systems benchmark method

Jia-Lang Seng\(^{a,}\)*, S. Bing Yao\(^{b,1}\), Alan R. Hevner\(^{c,2}\)

\(^{a}\)Deparment of MIS, National Chengchi University, Wenshan District, 116, Taipei, Taiwan, ROC
\(^{b}\)XBD System Inc., Columbia, MD, USA
\(^{c}\)Department of IS and DS, University of South Florida, Tampa, FL, USA

Received 31 January 2003; accepted 30 June 2003
Available online 27 January 2004

Abstract

Benchmarks are the vital tools in the performance measurement, evaluation, and comparison of relational database management systems (RDBMS). Standard benchmarks such as the TP1, TPC-A, TPC-B, TPC-C, TPC-D, TPC-H, TPC-R, TPC-W, Wisconsin, and AS\(^3\)AP benchmarks have been used to assess the performance of relational database management systems. These benchmarks are synthetic and domain-specific. Test results from these benchmarks are estimates of possible system performance for certain pre-determined application types. Database system performance on actual database domain may vary significantly from those in the standard benchmarks. In this paper, we describe a new benchmark method that is computer-assisted and developed from the perspective of the user’s requirements.

© 2003 Elsevier B.V. All rights reserved.

Keywords: Database benchmark; Performance measurement and evaluation; Relational database management systems; Requirements analysis; Workload characterization; Workload model; Synthetic benchmark

1. Introduction

A database benchmark is a standard set of executable instructions that are used to measure and compare the relative and quantitative performance of two or more database management systems through the execution of controlled experiments. Benchmark data such as throughput, jobs per time unit, and the inverse measure, response time, time per job unit, and other independent measures such as price performance ratio, equivalent database size, and Web interactions per second serve to predict and profile the system performance. It in turn assists a variety of user groups to make a decision on the system procurement, the capacity planning, and the bottleneck detection [11,26–28].

Database benchmarks comprise test databases and test workloads. They can be synthetic or empirical. Synthetic benchmarks emulate typical applications in a pre-determined problem domain and create a corresponding synthetic workload. Empirical benchmarks utilize real data and tests and re-invent the actual database applications. Though real workloads are ideal tests for systems, the costs of implementation of the actual environments usually outweigh the benefits provided by empirical benchmarks. Synthetic bench-
marks are therefore the common approach chosen by vendors and users. However, current synthetic benchmarks are mostly domain pre-determined. Reproduction of test results is not guaranteed and variation can occur at the actual user settings. As Refs. [7,8,11,25] pointed out in their works that, although standard and synthetic benchmarks model typical applications, there is no proven way to assess the representative degree of the typical applications. Users must be aware that test results may not be reproduced in their environment. Users are advised to revise and re-execute these experiments. Standard benchmark results are rough estimates and only serve the purpose of relative comparison. Simulated workload varies and domain changes. The degree of variation depends on the approximation of the workloads in the real environments to those defined in the standard synthetic benchmarks. No interpolation of standard results is recommended. Refs. [10,12,18,29,30] also point out in their works that the use of pre-determined workloads is the root of the structural inadequacy of synthetic benchmarks. Domain dependency and application bound deters the further enhancement of synthetic benchmark. And, the structural deficiency causes the irreproduced and irrepresentative performance results. At present, standard benchmarks consist of:

- The TP1 benchmark [1,17], the TPC-A, and TPC-B benchmarks [19–21] are on-line transaction processing (OLTP) benchmarks simulating one bank transaction type, later retired by TPC in 1995.
- The TPC-C benchmark is a complex OLTP benchmark emulating order entry and inventory control transactions in a production environment to replace TPC-A and TPC-B benchmarks [15,16,21].
- The TPC-D benchmark, later revised and replaced by the TPC-H and TPC-R benchmarks, is a complex on-line analytic processing (OLAP) benchmark simulating decision support systems (DSS) transactions [22–24].
- The TPC-W benchmark is the latest TPC official benchmark and an electronic commerce (EC) benchmark to model the Internet bookstore transactional workloads [25].
- The Wisconsin benchmark is a relational query benchmark [2–6].
- The AS3AP benchmark is a complex mixed workload benchmark [18,26–28].

These are standard relational database management systems (RDBMS) benchmarks. They use synthetic, domain-specific, and simulational workloads. They provide relative measurement and evaluation of system performance from pre-set profile. The key drawback is when the user domain differs from the standard domain and when the application workload deviates with the test workload, they cannot reflect the differences and changes in their design. Test results vary at user’s domain and mislead user’s decision.

In this research, we tackle the issue by proposing a domain-independent and application-independent synthetic workload model that is developed from the perspective of the user’s requirements. This synthetic benchmark method differs from the current standard benchmarks in several aspects. First, it is a computer-assisted synthetic benchmark environment where the benchmark development is automated as much as we can. It is a benchmark to be perceived as a workload formulation process of requirements representation, transformation, and generation in an automated manner. We adopt the concept of requirements analysis to be the carrier to capture and compile user’s requirements. The adoption is accomplished via the framework development from workload characterization. Data model, transaction model, and control model compose the carrier framework. Data analysis, transaction analysis, and control analysis apply the concept of requirements analysis. In specific, a high-level specification language, a translator of the language, and a set of generators are created to compose test databases and test transactions. User-friendly factor is the key concern to be a cost-effective design. Users enter their workload characteristics via commands, selects, and fill-ins. Translator automatically parses and produces the workload generation codes. Embedded SQL scripts and high-level programming statements are turned out. Test driver and result collector work with database generator and transaction generator to make test databases and issue database transactions against the system under test. In this study, we have conducted several benchmark experiments to test the method. We select the TPC-A, Wisconsin, AS3AP, and TPC-C benchmarks as our standard test cases in the first set of experiments. We create a more generalized and extended case in the second set of experiments to show the generality of our method. Experimental results including the produced workload
scripts, the generated databases, the issued test suites, and the collected timing statistics show the same or similar pattern and trend with those in the original standard benchmarks. These have proved the method is a superset of the current relational benchmark and a more generalized benchmark in modeling ad-hoc and requirements-driven workloads.

This paper is organized into five sections. Section 1 introduces this research. Section 2 presents the requirements-driven database benchmark. Section 3 describes the standard benchmark experiments and results. Section 4 details the extended benchmark experiment and the discussion of the generality of the method. Section 5 concludes this paper with a brief summary and future research directions.

2. Requirements-driven benchmark method

Fig. 1 presents the conceptual structure of the requirements-driven database benchmarking in a flowchart. It describes the association of the benchmark method with the overall process of generating and executing a benchmark. The user’s requirements are input to this process. These are represented in the workload modeling language. Workload requirements are transformed into program codes to generate the test database and the test transactions. A test driver executes the benchmark and collects the timing data from the experiments.

Fig. 2 presents the details of the requirements-driven benchmark method with its model components. First, we gather workload requirements in an entity relationship diagram (ERD) and a transaction process chart (TPC). They are modeled and formalized in the workload specification language. We translate the workload specification through a set of lexical rules and a set of syntactical rules to produce the parsed specifications, including the database specification, transaction specification, and control specification. This process is computer-aided with a user-friendly interface of fill-ins, selects, and commands. We then deploy the database generator with the database specification and create the test databases. We further apply the transaction generator with the transaction specification and issue the test transactions. We use the control generator with the control specification and produce the control scripts to support the test driver and collect the timing statistics. We follow the fundamental design property of a benchmark to be relevant, portable, scalable, and acceptable. In order to support our claim and to provide readers an opportunity to judge if the installation of our method is doable and reasonable, we spend the space of the following sections describing each of the components in detail.

2.1. Workload requirements gathering

A workload is the amount of work assigned to or performed by a worker or unit of workers in a given time period. The workload of a database benchmark is the amount of work assigned to or performed by a database system in a given period of time. Workloads can determine the scope and scale of the benchmark. Database workloads are best described by the amount of work, the rate at which the work is created, and the...
characteristics, distribution, and content of the work. Conventionally, workload modeling and characterization start with the domain survey, observation, and data collection, and continue with a study of the main components and their properties. Specifically, the database benchmark workload components include data, queries, transactions, and control. The workload characteristics include the description of type, volume, usage, and content of each component [9,10,13,14].

Workload modeling can be seen as requirements collection and analysis of the desired database applications. We model the system state and transactions that the database applications must use and perform. These requirements distinguish themselves as a specialized subset of systems requirements that concern databases, benchmark experiments, and test suites. Workload analysis involves data analysis and transaction analysis. In data analysis, we analyze data in terms of (1) the size of the database, (2) the number of records, (3) the length of the records, (4) the types of fields, (5) the value distributions and correlations, (6) the keys and indexing, (7) the hit ratios, (8) the selectivity factors, and (9) the joining fields and tables. In transaction analysis, we examine transactions in terms of (1) the complexity of transactions, (2) the correlation of transactions, (3) the input into the transactions, (4) the fields and tables used by the transactions, (5) the results size, and (6) the output modes. These are further investigated with the control requirements in terms of (1) the duration of test, (2) the steady state, (3) the number of users, (4) the precedence of transactions, (5) the frequency and distribution of jobs, and (6) the performance metrics desired. The collection and compilation of these database requirements

Fig. 2. Requirements-driven database benchmark method.
and workload characteristics formalize the framework of carrier of the data model, the transaction model, and the control model in the new method.

2.2. Workload specification language

Fig. 3 presents a block diagram of the main segments of the workload specification language. We divide the language into two parts. One part is the benchmark control segment that concerns the overall execution of the benchmark. The other part is the workload definition segment that contains the data and transaction definitions. A benchmark may consist of single or multiple workloads. Each is specified with benchmark control and workload definition. The workload definition is further divided into three segments. One segment comprises the database schema. The other segment includes the control information of the transaction generation. Another segment contains the transaction processes. A workload may have one or more transaction types. Each transaction is described with one set of transaction control and transaction detail segments. These together specify the workload requirements in a series of blocks of statements that are written in a formal grammar.

2.2.1. Benchmark control

The benchmark control segment describes the test duration of the benchmark, the precedence between transactions, the distribution of transactions, the number of tables and the number of transactions involved in the benchmark, and the steady state requirements of benchmark execution. Table 1 presents the set of syntax rules of the benchmark control segment. The syntax rules are stated in Backus-Naur Form (BNF) notations. To describe the test duration, we use the units of time or volume of transactions run in the benchmark. To specify the transaction dependency, we associate the related transactions with a key word, “before”. We define the transaction distribution using the syntax of distribution description. We give the unit of time in seconds to record the steady state requirements. We need the information as to the number of tables and the number of transactions in the computation procedures of the generation processes.

2.2.2. Data definition

The data definition segment contains the description of the database schema. We use three sections to record the information required to create tables, create indices, and load the database. Each section is headed with the key word of the function. The first section is “create”, which delineates the tables’ and fields’ definitions. The second section is “index”, which includes the query statements to create indices. The third section is “load”, which narrates the essence of the schema for data loading. Each section is appended with the query statements to execute the creation and insertion processes. Timing statements, such as ftime(&timebuffer1) and ftime(&timebuffer2), are added to measure the execution time of the operations. Table 2 presents the syntax of the data definition segment. We record the table name, the table size, and the number of fields for each table specified. We
use eight characteristics to describe the fields, including (1) the field name, (2) the field type, (3) the data length, (4) the data distribution, (5) the key and (6) index assignment, (7) the fixed-or variable-length size, and (8) the permission to be null or not. There are 11 data types, including smallint, longint, float, double, decimal, character, money, date, time, timestamp, and variable length character string for choices; four key types, including primary key, foreign key, secondary key, and non-key types; and three index types, including clustered index, non-clustered index, and no index for descriptions.

2.2.3. Transaction control

The transaction control segment describes the inter-arrival time distribution of the transactions, the input data definition, the number of inputs and the number of repetitions for the transactions. We present the frequency distribution using the distribution description. We portray the transaction input definition with the input name, the data type, the data length, and the input distribution description. The information of the number of inputs and the number of repetitions is required in the computing procedures. We give the syntax of transaction control in Table 3.

2.2.4. Transaction detail

The transaction detail segment contains the transaction statements to carry out the processes of each transaction type. As illustrated in Table 4, a transaction statement can be a query statement or a procedural statement such as an assignment, if, while, or I/O statement. The ANSI SQL syntax is used to write the query statements and a subset of the ANSI C syntax is adopted to express the procedural statements. The subset includes the digits, characters,
operators, expressions, conditions, assignment statements, if statements, and while statements syntax. The SQL statements are marked with the symbol, “#”, to be passed to the database systems. The timing statements are added to appraise and acquire the execution time of each transaction.

2.2.5. Distribution description

We detail the distribution descriptions in this section to explain the syntax of the probability distributions we use in the data, transaction, and control definitions. The detailed description serves to show the portability and scalability of our design. We support eight distribution descriptions, including (1) the fixed (or constant), (2) the uniform, (3) the exponential, (4) the rotating, (5) the normal, (6) the zipf, (7) the discrete, and (8) the sequential distributions. Table 5 presents the syntax of each of the distribution descriptions. The fixed distribution is used to record constant data. It is applied in three areas, including the transaction type and frequency distribution, the field value distribution, and the input data distribution. The uniform distribution is utilized to create data randomly and uniformly. The range of random values has to be entered for calculation. It is applied in the transaction, field, and input data definitions. The discrete distribution is used to define the transaction type distribution. It places different weights on the random numbers to be generated. Each transaction defined is given a probability of frequency of occurrence in the description.

The exponential distribution description specifies the exponential transaction frequency distribution. The rotating distribution delineates the extent of values to be generated alternately and uniformly within a circle. We can specify an increment such as 1 or 2 and odd or even to vary the values of adjacent numbers. It is used in the field definition. The normal distribution asks for the upper and lower limits of values to derive normal data. The zipf distribution requires the information, including the rank of the frequency of a value, the number of records to be generated with the value, and the value itself. Both distributions are employed in the field definition. And finally, the sequential distribution is used to describe the transaction type distribution.

---

Table 3
Transaction control syntax

```xml
<transaction_control_block>
 ::= transaction_control <transaction_control_id>
   { }
   frequency_dist:  <dist_text>; 
   input_dist:  *{<input_id> <data_type> <number> <dist_text>;}
   no_of_inputs: <number>;
   no_of_repetitions: <number>;
}
```

---

Table 4
Transaction detail syntax

```xml
<transaction_statement>
 ::= <assignment_statement>
   |<sql_statement>
   |<if_sql_statement>
   |<while_sql_statement>
   |<io_statement>
<assignment_statement>
 ::= <identifier> = <expression>;
<sql_statement>
 ::= sql;
<if_sql_statement>
 ::= if (<condition>)
   {<transaction_statement>}
   [else {<transaction_statement>}];
<while_sql_statement>
 ::= while (<condition>)
   {<transaction_statement>};
<io_statement>
 ::= read <identifier> from <device_type>;
   |write <identifier> to <device_type>;
<condition>
 ::= <expression> [<logical_operator> <expression>]
<expression>
 ::= <term> [<arithmetic_operator> <expression>]
<term>
 ::= <literal>
   |<number>
```
Transactions can be specified in any one by one sequential order.

2.3. Language translator

The language translator is developed to parse the formal grammar of the specification language. The lexical rules are made up of the key words of the language to produce the lexical tokens (or terminals). The syntactical rules are made up of the grammatical rules and actions to produce the parsed output. When a workload specification is parsed, the translator checks syntax, matches rules, executes actions, and creates generation specifications, including the control, the database, and the transaction generation specifications.

2.3.1. Control specification

To condense the description, we depict only the essence of the control specification. The control specification contains the control code derived from the control information and the characteristics information in the sections of benchmark control, data definition, and transaction control. The header and trailer sections are for the programming implementation use. The control data is converted into INSERT query statements to be stored in a set of control tables, which we categorize as: overall benchmark control, precedence control, field value distribution control, data generation control, data loading control, transaction input distribution control, and transaction generation control.

Fig. 4 presents the database structure of the main control tables. The test driver and the generators to setup the experiments will use each control table. In a workload, the main table, b_control_1, comprises the benchmark control requirements except the precedence definition, which is contained in the table, b_control_2. A workload may have zero, one, or more transaction dependency conditions. The cardinality
2.3.2. Database specification

The database specification contains the data and index generation code translated from the “create”, “index”, and “load” definition sections. It is composed of three files, the table creation file, the index creation file, and the data-loading file. The table creation file includes the query statements to create tables and the timing statements to measure the execution time. The index creation file contains the query statements to create indices and the timing statements to measure the execution time. The data-loading file includes a host variables declaration section and a host variables values assignment section.
to assign values to fields declared as host variables. The declaration and assignment sections permit us to create and assign new field values without changing the insert variable names. Each table to be loaded has a segment in the file.

2.3.3. Transaction specification

The transaction specification comprises the transaction generation code translated from the transaction detail section. It consists of a host variables declaration section and a host variables values assignment section for each input in the transaction, in addition to the transaction statements and the timing statements. Each transaction defined has a segment in the file.

3. Benchmark experiments

We developed a prototype of the method as an implementation. We used the prototype to conduct a set of standard benchmark experiments and a set of more generalized user experiments. These experiments were conducted as a proof of the generality and effectiveness of the new method. The prototype was constructed with a user interface to enter requirements, a language parser to parse specifications, and a set of generators to produce experiments. The user interface has been designed in a user-friendly manner and with fill-ins, selects, and commands to minimize the script writing and program coding efforts. The language parser has been written in the Unix lex and yacc to be platform portable and scalable. It was designed to parse and transform the user requirements into the database specification, the transaction specification, and the control specification in the form of SQL scripts and high-level programming statements. The database generator and the transaction generator will use these outputs and produce the test databases and issue the database transactions against the system under test. These generators have been written in the ANSI C codes and the embedded SQL codes. A comprehensive simulation program has been built to create the test driver so that it can drive and direct the benchmark experiments. Result collector is used to collect the timing statistics into the result database. We can analyze the performance results from it. The system runs on the IBM DB/2 platform. We show the system structure of the prototype in Fig. 5.

3.1. Wisconsin benchmark experiment

3.1.1. Experimental specification

The Wisconsin benchmark is a relational query benchmark. We specified the experiment using the single-user benchmark description described in Refs. [7,8]. The benchmark workload consisted of 32 query types, including selection, join, projection, aggregate, and simple update queries. The database contained four generic relations of onek, twok, fivelk, and tenk. We added five relations to duplicate the tenk relation for the query alternation use. Each relation comprised 16 attributes with 13 small integer numbers and three 52-byte character strings. The record length was 182 bytes.

There were five clustered indices built on the database. Three were created on the primary keys of the onek, twok, and fivelk relations. Two existed on the primary keys of two of the alternate tenk relations. Another two non-clustered indices were made on the candidate keys of the other two of the alternate tenk tables. We used the uniform and rotating distributions to create the fields’ values.

Table 6 shows a partial segment of the experimental specification. An important feature was the sequential run sequence specified for the 32 test queries. We ran the experiment for a period of 72 h. In the transaction definition, 10 variants of each of the selection, insert, update, and delete query types were used and specified. Four variants of each of the join, aggregate, and projection query types were described. We show the first query type, i.e. the 1% selection query without index, in the detail section.

3.1.2. Experimental results

The experimental results demonstrate that our method can produce the Wisconsin benchmark from the specification. Fig. 6 shows the workload created in the experiment. The 32 query types were generated and scheduled continuously in the specified run sequence by the transaction generator. In addition, our benchmark generated results similar to the Wisconsin benchmark. The test results of the selection queries with three indexing options changed as the selectivity factor varied. It illustrates the facts that the elapsed time increased with the results size and the use of index improved the query processing time. The test results of the join queries with three indexing options...
changed as the query complexity increased. It shows that the elapsed time increased with the join complexity and the use of index bettered the join processing time. The results of the aggregate queries using two indexing methods changed in the cases of MIN scalar, MIN functional, and SUM functional aggregates. The improvement of response time was once again achieved with the use of index. The results of the update queries using two indexing methods changed for the insert, delete, and update queries. The query to update the key field used more time than the others did. A possible reason was the extra time used to search and update the index and key files.

3.2. AS3AP benchmark experiment

3.2.1. Experimental specification

The AS3AP benchmark is a complex relational query benchmark. The experiment was specified using the benchmark description described in Refs. [26–28]. The benchmark workload consists of single-user tests and multi-user tests. The single-user workload comprises 39 tests, including the operational, selection, join, projection, aggregate, and bulk update queries. The multi-user workload contains four modules of tests, including the information retrieval (IR) test, the OLTP test, the mixed IR test, and the mixed OLTP test.
The AS3AP database includes four generic relations of unique, hundred, tenpct, and updates. Each relation has 10 fields with data types, including long integer numbers, double precision floating point numbers, decimal numbers, date, and character strings. On average, the record length is 100 bytes. Using the

![Diagram](image-url)

Fig. 6. Wisconsin experimental results—test workload generation.
given scaling equation, we calculated the database size and created the test database with 100,000 records for each relation. The number of users for the multi-user tests was computed to be 10.

There were four clustered indices built on the database. They were created on the primary keys of the four relations. A composite index was built on the tenpct table. Thirteen non-clustered indices were constructed. Four existed on the candidate keys of the four relations. Six were created on the tenpct relation and another three were created on the updates relation. We used the rotating, normal, and zipf distributions to generate the fields’ values. Table 7 presents a partial segment of the experimental specification. The single-user experiment was run for 12 h. The run sequence of the 39 queries was specified as required. Each of the

---

Table 7
A segment of the AS$^3$AP benchmark experimental specification

```plaintext
benchmark as$^3$ap
{
    benchmark_control 1
    {
        workload 1
        {
            duration: unit time 12 hour 0 minute 0 second;
            type_dist: sequence 5 9 8 6 7 19 13 4 27 26 12 20 14 29 28 30 11 16 18 15 21 24 22 25 10 17 23 11 33 38 32 34 36 35 37 39;
            steady_state: steady 0 hour 5 minute 0 second;
        }
        workload 2
        {
            duration: unit time 0 hour 20 minute 0 second;
            type_dist: constant 1;
            steady_state: steady 0 hour 5 minute 0 second;
        }
        workload 3
        {
            duration: unit time 0 hour 20 minute 0 second;
            type_dist: constant 1;
            steady_state: steady 0 hour 5 minute 0 second;
        }
        workload 4
        {
            duration: unit time 0 hour 20 minute 0 second;
            type_dist: constant 1;
            steady_state: steady 0 hour 5 minute 0 second;
        }
        workload 5
        {
            duration: unit time 0 hour 20 minute 0 second;
            type_dist: constant 1;
            steady_state: steady 0 hour 5 minute 0 second;
        }
    }
    ... transaction 5
    {
        transaction_control1
        {
            frequency_dist: null;
            input_dist: null;
        }
        transaction_detail 1
        {
            execute "select * from tiny";
        }
    }
    ...
}
```
multi-user tests was defined as an independent workload to be stated with the test interval, distribution, and steady state requirements.

3.2.2. Experimental results

The experimental results show that our method can produce the AS$^3$AP benchmark from the specification. Figs. 7 and 8 illustrate the experiment generation. These exhibit the generation of the 39 single-user query types that were scheduled continuously in the required run sequence by the transaction generator. These show the generation of the multi-user tests that were enforced through the transaction generator. There were 4 test modules, including 10 users running the pure IR test, 10 users running the pure OLTP test, 9 users running the pure IR test with 1 user executing the mixed IR test, and 9 users running the pure OLTP test with 1 user executing the mixed OLTP test in the background.

Our benchmark can generate results similar to those of the AS$^3$AP benchmark as in the Wisconsin experiment. The test results of the selection queries changed with two indexing methods as the selectivity factor varied. It illustrates the facts that the elapsed time increased with the selectivity factor and the response time decreased with more powerful index used. The test results of the join queries changed using two indexing methods as the join complexity varied. It shows that the more complex the join query was the longer the response time was. It exhibits the use of index could better the join performance. The three-way join query with clustered index was slower than the three-way join with non-clustered index. A possible reason was due to the introduction of a third table of tenpct that had extra non-clustered indices to help the case of join using secondary index.

Fig. 7. AS$^3$AP experimental results—single-user test workload generation.

Fig. 8. AS$^3$AP experimental results—multi-user test workload generation.
4. A more generalized experiment

4.1. Experimental specification

We developed a more generalized case as an attempt to show that the benchmark method can model generic workload requirements. We used the experiment to illustrate more realistic workload characteristics including the transaction distribution with probabilities, the transaction dependency condition, the input data requirements, and the use of clustered and character index, and the use of floating point index on the field which was not uniformly distributed, in addition to the use of regular integer indices. We reused the AS3AP database and altered the unique table to build new clustered index on the field, code, which was a character string, and a new non-clustered index on the field, float, which was a floating point number and in the zipf distribution.

Table 8 presents a partial segment of the generalized experimental specification. There were 21 transactions in the experiment. Each was given a probability of occurrence to specify the transaction distribution. There were two subsets of the transactions having transaction dependency and they were specified in the precedence line. We coded 21 transactions to include the workload characteristics stated above. They are described as follows.

(1) This transaction is an OLTP transaction that updates the database intensively. It comprises three processes. One process contains four one-row inserts into four tables with new input values. One process includes a while iteration to update the same table a number of times according to the input value. The other process is a conditional query that updates the database when the condition of input is met. This transaction has dependency with the second and third transactions, i.e. T1 precedes T2 and T2 precedes T3.

(2) This transaction is a DSS transaction that uses the results from the first transaction. It comprises two processes. The first process is a small report to list the values of a currently updated column from a table in order. The second process produces a larger report by joining two newly updated tables.

(3) This transaction is another OLTP transaction that is executed after the second transaction in order not to update the records before the reports can be produced. The transaction is to update a frequently accessed table using new input data.

(4)–(9) These transactions are to retrieve records whose values are between the range of character values using the primary keys; to retrieve records whose values are below or equal to a real number using the secondary keys; and to retrieve records whose values are below or equal to an integer number without using any keys.

(10)–(15) These transactions are to produce a small report from two currently updated tables using the primary keys that are character strings; to produce a small report from two newly modified tables using the secondary keys; and to produce a small report from the same tables without using any keys.

Table 8

<table>
<thead>
<tr>
<th>benchmark general</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmark_control 1</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>workload 1</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>duration: unit time 2 hour 0 minute 0 second;</td>
</tr>
<tr>
<td>precedence: 1 before 2; 2 before 3; 19 before 20; 20 before 21;</td>
</tr>
<tr>
<td>type_dist: weight 1 :: 0.3, 2 :: 0.2, 3 :: 0.1, 4 :: 0.03, 5 :: 0.03, 6 :: 0.03, 8 :: 0.03, 9 :: 0.03, 10 :: 0.03, 11 :: 0.03, 12 :: 0.03, 13 :: 0.03, 14 :: 0.03, 15 :: 0.03, 16 :: 0.01, 17 :: 0.01, 18 :: 0.005, 19 :: 0.005, 20 :: 0.005, 21 :: 0.005</td>
</tr>
<tr>
<td>steady_state: steady 0 hour 10 minute 0 second;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
(16) This transaction is to produce a list of the distinct values of every column in a table.

(17) This transaction is to count how many distinct values of a character key field.

(18) This transaction is to produce a larger report with averages, maximums, and minimums in groups and in orders using the secondary keys.

(19)–(21) These transactions emulate the processes of backup and recovery. One process saves a range of records according to the backup criteria. One process deletes the saved records. The other process recovers the deleted records from the backup table. These processes have transaction dependency that T19 precedes T20 and T20 precedes T21.

4.2. Experimental results

The experimental results show that our method can model general, ad-hoc, and custom workload requirements from the specification. Fig. 9 illustrates the generation of the experiment. The 21 transactions were produced and scheduled continuously by the transaction generator as the specified distribution required. In addition, the test results demonstrate a consistent trend of performance with the test complexity as discussed in the Wisconsin and AS3AP experiments. The results of the selection queries changed with increased selectivity factor. The response time increased as the results size increased and it decreased as the clustered index was used. The results of the join queries changed with increased join complexity. The elapsed time increased as the complexity increased and it decreased as the clustered index was applied.

4.3. Analysis of the experimental study

Table 9 presents the entire collection of the experimental factors and performance metrics we used in the more generalized benchmark experiment. They represent an extensive experimental study we have conducted to validate the requirements-driven benchmark method. There are 13 experimental factors, including (1) the database size, (2) the field data type, (3) the field value distribution, (4) the output mode, (5) the results size, (6) the selectivity factor, (7) the indexing method, (8) the query complexity, (9) the transaction type distribution, (10) the transaction frequency distribution, (11) the transaction dependency, (12) the input value distribution, and (13) the number of users. Table 9 indicates that our experiments intend to simulate a comprehensive benchmark workload in terms of test database and test transaction.

- The test databases: There are four database structures. Seven table sizes of 1, 10, 1000, 2000, 5000, 10,000, 100,000, and 1,000,000 records are defined. Three record lengths of 100, 108, and 182 bytes are used. Five numbers of indices that may be created in a table, for instance, zero, one, two, five, or eight indices per table, are specified. Six data types including short integer number, long integer number, floating point number, decimal number, date, and character string are used. Six data distributions including the constant, uniform, rotating, normal, exponential, and zipf distributions are applied.

- The test workloads: There are 13 query and transaction types. Queries can be operational,
selection, join, projection, aggregate, simple, or bulk updates. Transactions can be IR, OLTP, OLAP, DSS, or mixed. They can be queries with control logic, with dependency, or with input data. Transactions can be distributed constantly, sequentially, or weightily. In total, there are over ninety tests generated in the experiments. Three control constructs are designed into the workload including sequence, alteration, and iteration.

In addition, the experimental results show that our method can model the workload requirements from the user’s applications and our method can produce the consistent benchmark results. Comparing with the

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Description</th>
<th>TPC-C</th>
<th>Wisconsin</th>
<th>AS³AP</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectivity factor</td>
<td>Relation</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Absolute size</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Relative size</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Indexing method</td>
<td>Clustered index</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Non-clustered index</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>No index</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Data type</td>
<td>Integer</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Floating point</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Decimal</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Date/time</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Data distribution</td>
<td>Constant</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uniform</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rotating</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Discrete</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zipf</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exponential</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query type</td>
<td>Selection</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Join</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Projection</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Aggregate</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Simple update</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Bulk update</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Operation</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>DSS</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>OLTP</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Mixed</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>While</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequence</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction type</td>
<td>Constant</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>distribution</td>
<td>Uniform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction frequency</td>
<td>Constant</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>distribution</td>
<td>Exponential</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction dependency</td>
<td>Input data</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of users</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Single-user</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Multi-user</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experimental metrics</td>
<td>Elapsed time</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Throughput</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
original Wisconsin, AS3AP, and TPC-C benchmarks, we tested the method query by query and table by table. Figs. 6–8 show part of each of the generated workloads. Tables 6 and 7 show part of each of the produced specification. Figs. 10 and 11 show part of the timing statistics collected over these experiments. These all show either a same or a similar result and trend was reported in the original benchmarks. With the extended experiment, we try to emulate an ad-hoc relational workload including select, project, join, aggregate, bulk update, DSS-type query, OLTP-type query, with different selectivity factor, indexing method, output mode, test mode, and execution sequence. Experimental results show the method can model the ad-hoc generic benchmark as well as the standard synthetic benchmarks.

5. Discussion and summary

In this paper, we have described a requirements-driven synthetic benchmark method for relational database management systems. The method is developed to address the issue of domain dependency and workload dependency of current synthetic benchmarks. We have presented a detailed approach to model workload requirements from user’s perspectives. The intent is to provide a more generalized,
realistic, and computer-aided benchmarking process of workload representation, transformation, and generation. Generic data model, transaction model, and control model are developed to capture and compile workload requirements. In this study, we have conducted several standard and extended experiments of tests. Test results show that the method can model current standard synthetic benchmarks and user’s benchmarks. Experimental results show that the method is a more domain independent and workload independent synthetic benchmark. Reproducible and representative performance results were achieved.

Our future research will examine the expansion and extension of the method in the object-oriented and object-relational database systems over the Internet and distributed computing platforms. We continue to augment the system prototype to accommodate a larger scale of databases, more data types and data distributions, more complicated transaction processes, and a wider collection of performance metrics. The user interface will be enhanced to work with a visual requirements editor and writer. An expert system with performance monitor will be designed to analyze and pinpoint the system bottlenecks and tuning to provide benchmarking advice and action.

References

