AS3AP -
An ANSI SQL Standard Scalable and Portable Benchmark for Relational Database Systems

Carolyn Turbyfill
Cyril Orji
Dina Bitton*

1. Historical Perspective

In 1983, the first commercially available database machines were becoming available and new prototypes, such as the GRACE Database Machine [Kit83], had been proposed. The Wisconsin Benchmark [BDT83] was created in an attempt to compare backend database machine architectures to software relational database systems running on a general purpose computer. Specifically, the database machines tested were the Britton-Lee IDM 500 and Direct [Dew79], and the software database systems were University Ingres on Berkeley Unix and commercial Ingres on the VMS operating system [Ston86]. In attempting to compare these systems, we soon realized that an appropriate benchmarking methodology for comparing database system architectures was, in and of itself, a topic of research. Hence, the Wisconsin Benchmark was born as an experiment in comparative benchmarking methodology.

The Wisconsin Benchmark is a single-factor, controlled experiment using a synthetic database and a controlled workload. We expected, in the course of developing the benchmark, to encounter confounding variables that would require revision of our methodology. In order to develop a methodology for both benchmarking and analysis of results, we simplified the initial experiments, intentionally ignoring factors that we knew to be important such as attribute type, data distribution, database size, and number of users. We used only two data types: fixed length strings and two byte integers. All data was uniformly distributed. The database was 5 megabytes in size and the queries were not designed to scale with different size databases. The initial tests were performed with only one user on the entire system (single-user and standalone). The benchmark was also not designed to be extensible, although some extensions were possible [BT84, BD84, BHT87]. While our preliminary work on the Wisconsin Benchmark and variants of the benchmark did provide valuable insight into benchmarking methodology, the database and

* The initial version of this paper was written at the University of Illinois at Chicago. The authors current addresses are: Carolyn Turbyfill, Sun Microsystems; Cyril Orji, University of Illinois at Chicago; Dina Bitton, DB Software Corporation, Palo Alto, California.

queries as instantiated did not and were not intended to provide an adequate framework for comparative benchmarking.

Shortly after it was first published, the Wisconsin Benchmark became a defacto industry standard. With numerous relational database systems becoming commercially available, the need for an evaluation and comparison tool emerged. The simplicity of the Wisconsin benchmark design was a major contributing factor to its widespread use, as it was fairly easy to understand and implement. Another factor in the success of the benchmark was the widespread availability of numbers for different systems that provided a common point of comparison for a wide variety of architectures and implementations [BT88]. Even though some of the test queries had confounding variables such as predicate independence, which obscured the interpretation of the results [Tur87], and the test suite was not comprehensive, the Wisconsin benchmark was a very useful tool at the time it became available.

Currently, there is a need for a general benchmark that that can be used to compare relational database systems with vastly different architectures and capabilities over a variety of workloads. To achieve this goal in a cohesive and tractable benchmark, it is important that these criteria be central to the design of the benchmark, as opposed to being added as an afterthought to an existing benchmark. This was the motivation for the design of the AS^{3}AP benchmark [Tur87, TOB89].

2. A Scalable Benchmark

The benchmark described in this paper is an ***A***NSI ***S***QL ***S***tandard ***S***calable and ***P***ortable (***AS^{3}AP***) benchmark for relational database systems. It is designed to:

- provide a comprehensive but tractable set of tests for database processing power.
- have built in scalability and portability, so that it can be used to test a broad range of systems.
- minimize human effort in implementing and running the benchmark tests.
- provide a uniform metric, the *equivalent database ratio*, for a straightforward and non-ambiguous interpretation of the benchmark results.

For a particular DBMS, the AS^{3}AP benchmark determines an *equivalent database size*, which is *the maximum size of the AS^{3}AP database* for which the system is able to perform the designated AS^{3}AP set of single and multiuser tests *in under 12 hours*. The equivalent database size is an absolute performance metric by itself. It can also provide a basis for comparing cost and performance of systems, as follows: the *cost per megabyte* of a DBMS is the total cost of the

DBMS divided by the equivalent database size. The *equivalent database ratio* for two systems is the ratio of their equivalent database sizes. Both the cost per megabyte and the equivalent database ratio provide global comparison metrics.

Current relational database systems vary widely in their capabilities, performance, and cost. Thus the task of defining meaningful measures of database processing power is complex and the need for a scalable and portable benchmark is critical. There have been a number of previous attempts at formalizing and standardizing a benchmarking methodology for database systems [BDT83, Ano85, BT85, BHT87, RKC87, YHM87]. However previously proposed benchmarks fail to provide a comprehensive and uniform measure of performance. For instance, one provides a measure of transaction power for short on-line transactions [Ano85], while the other only tests features of the query optimizer [BDT83, BT88], and none of them test important utilities such as the database load and restore.

Benchmarks often require too much human effort to be implemented, ported, or realistically scaled to evaluate a particular system. Finally, few benchmarks are designed to facilitate the interpretation of their results and most fail to provide useful guidelines for fixing or improving the system under test. We have addressed these problems in the design of AS^{3}AP, by emphasizing scalability, portability, and ease of use and interpretation.

3. Benchmark Scope

The AS^{3}AP tests are divided into two modules:

- Single-user tests, including:
  (a) utilities for loading and structuring the database,
  (b) queries designed to test access methods and basic query optimization - selections, simple joins, projections, aggregates, one-tuple updates,and bulk updates.
- Multiuser tests modelling different types of database workloads:
  (a) on-line transaction processing (OLTP) workloads,
  (b) information retrieval (IR) workloads,
  (c) mixed workloads including a balance of short transactions, report queries, relation scan, and long transactions.

The set of single-user tests are designed as a cross-section of the basic functions that a relational DBMS must support, as defined by the ANSI SQL 2 Standard [ANSI90]. The multiuser tests establish the maximum throughput for OLTP, measure degradation in response time as a function of the system load, and provide measures of DBMS performance as a function of the workload

profile. In the multiuser tests, the mixed workload tests have been specifically designed to press the state of the art in OLTP [Tur88]. In combination with the equivalent database size metric, some of the single-user and multiuser tests are specifically designed to press the state of software maturity in parallel and distributed systems (see Section 6).

A basic tradeoff we confronted in the design of this benchmark was tractability versus completeness. The benchmark is not a test of SQL completeness and is not intended to model a specific workload or query mix. We have achieved tractability by setting the time limit for the benchmark at 12 hours. With respect to completeness, we prioritized the tests so that the most fundamental and important queries and workloads will be included in the 12 hour limit, as opposed to more specialized queries and workloads[1].

The AS^{3}AP database is designed so that special queries can be created to test: (i) correct optimization in the presence of non-uniform distributions, correlated attributes, and program variables; (ii) complex queries such as relational division, join-aggregates, recursive queries.

4. Systems under Test

Systems tested with the AS^{3}AP benchmark are assumed to support common data types, and to provide a complete relational interface with basic integrity, consistency, and recovery mechanisms. Otherwise, they may range anywhere from a single-user microcomputer DBMS to a high-performance parallel or distributed DBMS.

The only hardware normalization required is that the logical size of the AS^{3}AP database used for the benchmark be at least equal to the logical size of physical memory on the host(s). The logical size of the AS^{3}AP database is defined as the flat files storage requirements of the tables (see Section 5 below):

(# tuples per relation) x (100 bytes/tuple) x (4 relations in database)

This condition avoids biasing the computation of the equivalent database ratio in favor of systems that incur a high storage overhead for storing a database. It also makes testing main memory database systems [BT86, BHT87] with the AS^{3}AP benchmark meaningful, since the test database can be as large as physical memory.

[1] In previous work we have found that queries that take a few minutes when properly optimized can take hours when improperly optimized [Tur87]. Hence, for queries to be included in the 12 hour limit, we have endeavored to design queries that detect query optimization problems without completely dominating the benchmark if something goes wrong. For the same reason, aspects of query optimization, such as prediction of selectivities in precompiled queries with program variables, are tested only once and not in every possible query.

All systems, including micro DBMS's, should be tested with the first module, the single-user tests. All multiuser DBMS's should be tested with both modules, single-user and multiuser tests. For micro DBMS's, the equivalent database ratio is computed based on the single-user tests only. For other systems, it is computed based on the single-user and multiuser tests.

5. Measurements

The only measurement required by the AS^{3}AP benchmark is query elapsed time, within the 12 hour window limit. Other performance data on CPU and I/O utilization must be collected for in depth analysis of a DBMS performance. However, these additional measurements are not part of the AS^{3}AP metrics. All the AS^{3}AP queries composing a run are embedded in a host program, along with the required timing commands. All queries are precompiled, except for the multiuser cross section queries listed in Table 8.1 and Appendix 4. For the multiuser tests, a number of programs (processes) are forked concurrently, each running a simple script. The benchmark does not use a terminal emulator and does not generate streams of job arrivals. There are no think time or presentation services included in the measurements. Preliminary queries in the single-user tests isolate the cost of different output modes. For all the other queries, the results are discarded (e.g., redirected to /dev/null in Unix).

Because of these simplifications, the benchmark can be set up to run very easily. The tests are grouped into a number of modules, according to the component of the DBMS that they are testing. It is a strict requirement that all the basic tests (single and multiuser) run in a total time of less than 12 hours. The benchmark can be run as one large program, or as a set of independent modules if the host system is not available in standalone mode for a 12 hours interval of testing.

The database is not corrupted by the test suite as long as the user runs the complete suite in the specified order. This is achieved by interleaving special queries that save deleted tuples, reinsert the deleted tuples, and restore updated tuples to their original values

6. Test Database

6.1.Database Generator

The test database is generated using DBGEN [BMO88], a parameterized C-program, that provides random generators of numbers and character strings, with a number of common distributions (uniform, normal, exponential, zipfian[2]). It is parameterized so that the range, number of unique values, mean, standard deviation, and correlation factors can be specified.

Four flat character files are generated for AS^{3}AP. After these files are generated, they should be loaded into four relations in the DBMS under test. The first AS^{3}AP test consists of timing this load operation. The load utility varies widely from one system to the other. Some systems provide a simple file copy utility, while others include the cost of clustering the file and building indices.

All four relations have the same average tuple width and the same number of tuples. The tuple length is 100 bytes on the average. However, actual tuple lengths vary due to the variable length string attribute *address* which has a minimum length of two bytes and a maximum length of 80 bytes (Table 6.3).

The size of the four relations can be scaled from 1 megabyte to 100 gigabytes by varying the number of tuples from 10,000 to one billion. Table 6.1. summarizes the range of logical sizes for one relation and for the database. The physical storage required to store the database in the DBMS under test may be higher due to non standard implementation of data types and DBMS overhead.

Table 6.1.: Scaling the AS^{3}AP Database

relation size (tuples)	**relation size** (bytes)	**logical database size** (bytes)
10,000	1 megabyte	4 megabytes
100,000	10 megabytes	40 megabytes
1,000,000	100 megabytes	400 megabytes
10,000,000	1 gigabyte	4 gigabytes
100,000,000	10 gigabytes	40 gigabytes

[2] Zipfian distributions are a family of non-uniform distributions that have been showed to accurately model text data. In the Zipf distribution, for each value, the product of rank and frequency is a constant [Knu73]. Rank is an ordering according to frequency of occurrence. In a zipfian (or Zipf-like distribution) the product of a power of the rank and frequency is a constant. The power is called the decay factor. A decay factor of zero corresponds to the uniform distribution. The distribution becomes more skewed as the decay factor increases

1,000,000,000	100 gigabytes	400 gigabytes

As defaults for the test database, we provide a one megabyte base relation (for micros) and a ten megabyte base relation (for other systems). However, in order to get optimal rating of their DBMS on the benchmark, implementors must estimate the maximum size of the test database that will enable them to meet the 12 hour requirement.

This limit on the benchmark run time determines the *equivalent database size* for the DBMS under test. When choosing the size of their test database, implementors must balance two conflicting factors: the database size and the total time required to run the benchmark. In order to obtain shorter elapsed times for the test queries, they will want to have a smaller database. On the other hand, in order to compare favorably with other systems, implementors will want to run the benchmark with the largest possible database since the AS^{3}AP comparison metric is the *equivalent database ratio*.

6.2 Structure of the Database

The AS^{3}AP database contains five relations. One, the *tiny* relation, is a one tuple, one column relation, used only to measure overhead. The other are the four relations generated by the database generator, with the appropriate scaling. These relations are named as follows:

uniques. A relation where all attributes have unique values.

hundred. A relation where most of the attributes have exactly 100 unique values, and are correlated. This relation provides absolute selectivities of 100, and projections producing exactly 100 multi-attribute tuples[3].

tenpct. A relation where most of the attributes have 10% unique values. This relation provides relative selectivities of 10%.

updates. A relation customized for updates. Different distributions are used and three types of indices are built on this relation.

The four relations have the same ten attributes (columns) with the same names and types, and if needed to comply with tuple width specification, a *fill* attribute. The attributes (Table 6.2) cover the range of commonly used data types: signed and unsigned integer, floating point, exact decimal, alphanumeric, fixed and variable length strings, and an 8 character date type. The

[3] To support projections producing 100 single attribute tuples, it is sufficient to have an attribute with 100 unique values. In order to obtain 100 tuples by projecting on more than one attribute, we generated correlated attribute values so that projecting any subset of nonunique values also produces exactly 100 unique values.

precision of floating point numbers is within the limits of the IEEE Standard floating point number. Furthermore, only range predicates (i.e. float between 998 and 1000) are used with floating point numbers as precision differences make equality predicates (such as float = 999) less portable.

The attribute lengths sum up to 100 bytes if the DBMS support all the required data type with a standard length (i.e. 4 byte integers, 8 byte floating points, etc.) An additional attribute, *fill*, is provided to comply with the 100 byte tuple width requirement, when the DBMS does not support one or several of the data types as specified in Table 6.2. For systems that do not support the needed data types or do not store them efficiently, portability of the test database is enforced by providing precise substitution rules. For instance, if variable length character strings are not supported, the tenth attribute, *address*, should be a fixed length character string of 80 bytes, and the tuple width will be 160 bytes. Thus a DBMS that does not support variable length strings is penalized by higher storage requirements for the test database. If another data type, say exact decimal, is not supported and the implementor substitutes a shorter type for it, then the fill attribute should make up for the difference in storage requirement between the required type, exact decimal, and the substitute. Only integer values are generated by the test database generator so that all predicates on a numeric value will select the same values when a 4 byte integer is substituted for any numeric type.

Table 6.2: Attribute Names, Types, and Length in Bytes

Attribute Number and Name[4]	Type	Length
1. key	integer	4
2. int	unsigned integer	4
3. signed	signed integer	4
4. float	floating point	4
5. double	double precision	8
6. decimal	exact decimal	18.2
7. date	datetime	8
8. code	alphanumeric	10
9. name	character string	20 (fixed)
10. address	variable length string	2 to 80 (20 avg)
11. fill	char string	as needed
		total=100 (avg)

[4] Certain DBMS's do not allow using the same attribute name in 2 relations. In that case, we suggest prefixing the attribute names with their relation name.

The SQL schema for the test database is provided in Appendix 1, where the data types are as specified in Table 2 and the tuple width is 100 bytes. An example is also provided where substitution rules require a fill attribute with 6 characters.

In the test database, the values of every attribute are randomly generated, with a uniform or a non-uniform distribution. Table 6.3 contains a detailed description of the attributes, including their type, the range and distribution of their values, whether they constitute a key or a foreign key, the type of index built on the attribute, if any, and how many unique values they have. We describe indices as: clustered or nonclustered, primary or secondary, B-tree or hashed. Indices are specified on subsets of attributes, here called the *index key*, not to be confused with the primary key of a relation. In a *clustered index*, the index determines the physical placement of the data. For instance, in a *clustered B-tree index*, the data is sorted on the index key. In a *clustered hashed index*, the page or bucket a row is placed in is determined by a hash function on the index key. In a *nonclustered index*, the data is not sorted on the index key. The index is used to locate a pointer to a row or rows containing the index key value. In a *nonclustered B-tree index*, the leaves of the tree usually contain one instance of each unique index key paired with pointers to each row containing attributes corresponding to the index key value. In a *nonclustered hashed index*, a hash function on the indexed key is used to locate pointers to the desired row(s). A *primary index* is an index on the relation primary key. A primary index is usually a clustered index. A *secondary index* is an index on attributes other than the primary key. A secondary index is usually nonclustered.

We assume that three types of indices are supported by the DBMS under test:

(1) A *B-tree clustered* index (abbreviated *cl. btree* in Table 6.3), which, for all the relations, is built as the primary index on the *key* field,

(2) A *B-tree secondary nonclustered index* (abbreviated *sec. btree* in Table 6.3), and

(3) A *hashed secondary nonclustered index* (abbreviated *sec. hash* in Table 6.3).

We have one composite primary clustered index on the key and code attributes of the tenpcnt relation. Multiple secondary indices are built on the relations, as indicated in Table 6.3. A secondary hash index is built on the alphanumeric field *code* for all four relations.

With respect to physical database design, the following substitution rules apply:

- Any desired index can be substituted for a clustered B-tree index specified in Table 6.3. If available, a clustered index must be substituted instead of a nonclustered index.
- Any desired nonclustered index can be substituted for a nonclustered B-tree index specified in Table 6.3.

- Any desired nonclustered index can be substituted for the nonclustered hashed index specified in Table 6.3. If available, it must be different from the nonclustered B-tree index or the index substituted for the nonclustered B-tree.

Of course, systems that automatically invert or otherwise encode all attributes or tuples are allowed to do so.

The range of values for an attribute is determined by its type and by the size of the relation. For some numeric types, two types of ranges are included, *dense* and *sparse*. In a dense range, every integer value in the range is represented in the table. For instance, the *key* in the *updates* has the range of 0 to (# of tuples) in the *updates* table. In the 4 megabyte database, the *key* in the *updates* table has a range from 0 to 10,000. In the 400 gigabyte database, its range is from 0 to 1 billion. With all dense ranges, the value 1 is omitted to allow "not-found-scans" for a value within the range of attribute. These scans measure the rate at which data can be read and tested.

Dense ranges are used to phrase queries with *fixed selectivities* independent of the database size, such as the query **o_mode_100k**:

```
select * from updates where key <=100
```

which always selects 100 tuples. In a sparse range, the range is independent of the database size. For instance, the *key* attribute in the *uniques* relation is an integer with values in the range $0\text{-}10^9$, to allow for up to 10^9 distinct values. When the number of tuples needed is smaller than 10^9, the range is *sparse* (see Table 6.3.). Sparse ranges are used to phrase queries with *relative selectivities* that are a function of the database size, such as the query **sel_10pct_cl** :

```
select key,int,signed,code,double,name
from uniques
where key <= 100000000
```

which always selects 10% of the tuples in the uniques relation.

The range and distribution of attributes with the same name may be different in the different relations, in order to provide the functionality required by the queries. For instance, the range of the *key* attribute in *uniques* is 0 to 10^9 and it is sparse. In the *hundred* relation, the range for the same attribute is 0 to number of tuples and it is dense (except for the value "1" which is missing); And the *double* attribute has a normal distribution in the *uniques* relation but a uniform distribution in the *hundred* relation.

Ranges are defined for all ten attributes so that scaling of the database is straightforward, and the queries are not changed when the database is scaled. Once the number of tuples in the database relations is specified (10^4 to 10^9), the generator program will generate the appropriate attribute values. *The queries are designed to automatically scale with the databas*e. In particular, the

attribute ranges are specified so that the query selectivities remain unchanged and all the required constant values used in the predicates remain unchanged when the relation size is scaled.

Table 6.3. : Description of the Attributes of the 4 Test Relations					
attr. name	**attr. type**	**uniques**	**hundred**	**tenpct**	**updates**
1. key *prim. key*	integer(4)	0 - 10^9 1 missing sparse uniform *cl. btree*	0 - #tuples 1 missing dense uniform *cl. btree*	0 - 10^9 1 missing sparse uniform *cl. btree*	0 - #tuples 1 missing dense uniform *cl. btree*
2. int	integer(4) unsigned	0 - 10^9 1 missing sparse uniform	0 - 10^9 1 missing sparse uniform	0 - 10^9 1 missing sparse uniform *sec. btree*	0 - #tuples 1 missing dense uniform *sec. btree*
3. signed	integer(4) nullable	± 5•10^8 sparse uniform	100 - 199 dense uniform *foreign key for updates*	± 5•10^8 sparse uniform *sec. btree*	± 5• 10^8 sparse uniform
4. float	real(4)	± 5•10^8 zipf	± 5•10^8 uniform 100 uniques	± 5•10^8 uniform 10% uniques *sec. btree*	± 5• 10^8 zipf 10 uniques
5. double	double(8)	± 10^9 normal distn	± 10^9 uniform 100 uniques	± 10^9 uniform 10% uniques *sec. btree*	± 10^9 normal distn all unique *sec. btree*
6. decim	numeric(18,2)	± 10^9	± 10^9 100 uniques	± 10^9 10% uniques *sec. btree*	± 10^9 all unique *sec. btree*
7. date	datetime 8 bytes	1/1/1900 - 12/1/2000 uniform	1/1/1900 - 12/1/2000 uniform	1/1/1900 - 12/1/2000 uniform	1/1/1900 - 12/1/2000 uniform
8. code *cand. key*	char(10)	alphanumeric *sec. hash*	alphanumeric *sec. hash*	alphanumeric *cl. btree* *sec. hash*	alphanumeric *sec. hash*
9. name	char(20)	alphanumeric	alphanumeric 100 uniques	alphanumeric 10 uniques *sec. hash*	alphanumeric all unique

10. address	var char(80)	alphanumeric	alphanumeric 100 uniques	alphanumeric 10% uniques	alphanumeric all unique
11. fill	char(?)	alphanumeric	alphanumeric	alphanumeric	alphanumeric

7. Single-User Tests

Single-user tests are logically divided into operational issues and user queries. Operational issues include load, backup, building indices, and a check of referential integrity. Queries include retrievals, single-tuple updates, and bulk updates. All are run in standalone mode, and the elapsed time for each test or query is measured. In our single-user tests, as the database gets larger, systems will be penalized if they do not use parallel and distributed algorithms in the execution of utilities or large retrievals. On the other hand, some of the simple queries, that access only a few database pages, will be fairly insensitive to database size as long as relevant indices remain in memory. Our purpose is to identify systems that are truly capable of supporting large databases and processing large quantities of data. This is in contrast to the Debit-Credit benchmark, where systems which do not employ parallel or distributed algorithms in the execution of a single query can perform quite well, both in elapsed time and in cost/performance metrics. What the Debit-Credit benchmark does not show the user is the time required to load, backup, or restore the database, or how any query that required processing large amounts of data, such as a batch or report query, would perform. A system could perform the random accesses required to perform the Debit-Credit benchmark superbly, and still be completely inadequate when performing other functions expected of a relational database system.

Table 7 shows the amount of time required to access data by access type (sequential or random). By way of illustration, assume a relatively rapid time of 20 milliseconds for each random access to 8K bytes, and assume sequential accesses proceed at the rate of 3 million bytes per second. These are optimistic numbers for serial access to data at the application level. At 20 milliseconds per 8K of data, it would take 2.44 seconds to process a megabyte of data and 28.26 days to process a terabyte of data. With a sequential access rate of 3 megabytes per second, it would take 0.33 seconds to access one megabyte of data and 3.86 days to access a terabyte of data. Table 6 illustrates the fact that queries that process large amounts of data cannot be efficiently executed without the use of distributed and parallel algorithms for the execution a single query. This is referred to as intra-query parallelism, as opposed to inter-query parallelism where several queries are executed in parallel. In practice, when new parallel and distributed architectures are implemented, the first software ported to the distributed or parallel hardware supports inter-query

parallelism by dedicating a different CPU to each query. Systems capable of intra-query parallelism represent the current state-of-the-art in software maturity.

Table 7: Time to Access Large Amounts of Data by Access Type		
Table Size in Bytes	Random Access @8KB / 20 msec	Sequential Access @ 3 MB/sec
1 megabyte	2.44 seconds	0.33 seconds
1 gigabyte	40.69 minutes	5.56 minutes
1 terabyte	28.26 days	3.86 days

Below, the operational tests are described first. Then the queries, organized according to function, are described. The tests and queries expressed in SQL are listed in Appendix 2. The order in which Section 7 and Appendix 2 list the queries is not the order in which they are run in the benchmark. AS^{3}AP uses a particular ordering of the queries to prevent the data of one query to be memory resident as a consequence of the previous query. It addition, the query order is designed to facilitate memory residence of relevant indices when this is appropriate. A run-time sequence has been determined that differs from the logical organization presented below. The actual AS^{3}AP run sequence is shown in Appendix 3.

Queries and operational tests are interleaved so that each query will access a relation different from the one(s) accessed by the previous query. This avoids lengthy operations that would otherwise be needed to flush the buffers. The probability of cache hits in main memory is also reduced as the database gets larger.

7.1. Operational Issues

After the AS^{3}AP database is created, and the schema specified, the first measurement performed is the time to load the four data files generated
relations. The data files may be loaded from disk or tape. Then, one of the relations, the *updates* relation, is backed up to tape or other archive device and the backup time is measured. A recovery of the *updates* relation is included in the multiuser tests.

We measure the time to build both clustered and secondardy indices. Usually, the load utility will cluster the file with respect to the declared primary key and build the appropriate primary index. In the event that the load does not include this function, the time for building a clustered index should be measured. Building the required indices (clustered indices if not handled by the load, and secondary indices as needed by the test queries) is interleaved with the queries. In the *uniques*, *hundred*, and *updates* relations, the clustered index is on the *key* attribute. The *tenpct* relation has a composites primary clustered index on the *key* and *code* attributes. We measure the time for building two types of secondary indices. For example, we build a secondary B-tree

index on the integer attribute *int* of the *tenpct* relation; we build a secondary hashed index on the *code* attribute of each relations: *uniques, hundred, updates* and*tenpct* .

A table scan is included to test sequential I/O performance (**table_scan**):

```
select * from uniques where int=1
```

Since the value "1" is not present and no index is built on the *int* column, this query scans the entire table but retrieves no tuple. Partial table scans that retrieve 100 and 1000 sequential tuples to the screen or to a file are also included in the tests of output mode (see Section 6.2.1. below).

7.2. Test Queries

7.2.1. Output mode Queries

The first set of tests exercises the three different output modes that a query may use. Results of a query may be (i) retrieved to the screen; (ii) stored in a file; (iii) stored in a database relation. We measure the overhead associated with each of these three output modes, per query (selection on *tiny* relation) and as a function of the amount of data retrieved (1k, 10k, or 100k bytes)[5]. The output mode queries are briefly described in Table 7.1. Measurements for these queries and a comparison to the table scans provide a precise estimate for the time required by the DBMS to format, display, and store the result of a query. After this first set of tests, the results of all queries are discarded, so that a query elapsed time measures the effectiveness of query optimization independently of the output mode. In order to further normalize the measurements of query elapsed time, the width of the result tuple is kept equal to 50 bytes for most test queries.

Table 7.1.: Output Mode Queries.
Each Query Repeated with Output To Screen, File, and Relation

Query Name	Description
o_mode_tiny	selection on tiny relation
o_mode_1K	selection of 10 tuples
o_mode_10K	selection of 100 tuples
o_mode_100K	selection of 1000 tuples

[5] In this context, 1k=1,000 bytes exactly. 1k corresponds to ten average logical tuples. The actual data moved may vary with implementations.

7.2.2. Selections

The speed at which a DBMS can process a selection query depends primarily on the storage organization of the relation and on the selectivity factor of the query. These two parameters are controlled in our tests for selections. There are seven selection queries, as listed in Table 7.2. A selection query selects one tuple, or a *range* of 100 tuples or 10% of the tuples. For each of these three selectivities, the query comes in two versions: one version uses a clustered index, and the other uses a secondary B-tree index. Finally, one selection query, **variable_select**, is a range selection with a secondary B-tree index, where the range is defined by a program variable. This query tests the ability of the query optimizer to correctly choose between a scan or the use of an index at run time[6]. In Appendix 3, which specifies the run sequence of queries, the **variable_select** query is executed twice in a row, first with a low selectivity when a nonclustered index should be used to evaluate the selection predicate, and second with a high selectivity, when the relation should be scanned to evaluate the predicate. Putting the two queries right after each other detects whether program variables are properly handled without overweighing the benchmark with two unnecessary relation scans. If the first **variable_select** is improperly executed by scanning the entire relation, the second will be relatively fast because the table should be memory resident for the second scan.

Table 7.2.: Selection Queries

Query Name	Description
sel_1_cl	select 1 tuple using clustered index
sel_1_ncl	select 1 tuple using secondary hashed index
sel_100_cl	select 100 tuples using clustered index
sel_100_ncl	select 100 tuples using B-tree secondary index
sel_10p_cl	select 10% tuples using clustered index
sel_10p_ncl	select 10% tuples using B-tree secondary index
variable_select	range select predicated on program variable

[6] Three queries with program variables are included: **variable_select**, and the background queries in the multiuser tests, **ir_select** and **oltp_update** (Section 7 and Appendix 4). Other tests isolate other access methods or other aspects of query optimization. We do not wish to confound those tests with improper handling of program variables should it occur.

7.2.3. Joins

The join queries test how efficiently the system makes use of available indices and how query complexity affects the relative performance of the DBMS. There are eight join queries, differing in the number of relations that they reference and in the availability and type of index available on the join attribute (see Table 7.3.). Seven of the eight queries are 1:1 joins and produce one result tuple. One query, **Join_1_10**, is a 1:10 join of two relations with a non-clustered index on the join attribute, and it produces 100 result tuples. The join with no index (**Join_2**) constitutes a simple test of performance on ad-hoc queries. When no index is available, a query optimizer should never use a nested loops join algorithm, because of its n^2 complexity. Instead, it should sort or hash the operand relations on the join attribute(s), and then merge-join them in one pass. Thus, if correctly optimized, this query measures the efficiency of a basic access method - sorting or hashing; otherwise it will exhibit very poor performance.

Table 7.3. : Join Queries

Query Name	Number of Tables	Index Available	Tuples Retrieved
join_2	2	no index	1
join_2_cl	2	clustered index	1
join_2_ncl	2	non-clustered hashed index	1
join_3_cl	3	clustered index	1
join_3_ncl	3	non-clustered hashed index	1
join_4_cl	4	clustered index	1
join_4_ncl	4	non-clustered hashed index	1
join_1_10	2	non-clustered B-tree index	100

Use of the available indices in the join algorithm is tested with three types of indices: the primary clustered index, a secondary hashed index, and a secondary B-tree index. In each case the appropriate type of index is available on the join attribute in all the operand relations.

Query complexity is modeled by increasing the number of tables referenced from two to 4. While most query optimizers will use the correct access plan on 2-way joins with a clustered or a non-clustered index, often they will not correctly optimize three and 4-way joins because of the higher complexity involved in evaluating all the possible access methods.

7.2.4. Projections

Most of the processing time for a projection is incurred by the elimination of duplicate tuples introduced when projecting on non key attributes. This makes the cost of a projection much higher than the cost of a selection query with similar selectivity. Duplicate elimination is usually performed by sorting. Thus tests of projections also provide a test of the sort utility used by the DBMS.

There are two test queries for projections (Table 7.4.). One query projects the *hundred* relation on two columns, corresponding to a signed integer and a variable length character attributes, and it produces 100 result tuples. Thus it provides a test of how efficiently the DBMS handles two data types in a complex operation such as sorting. The second query projects on one column, corresponding to a signed integer attribute, and has a 10% selectivity.

Table 7.4.: Projection Queries

Query Name	**Description**	**Selectivity**
proj_100	project on *address* and *signed* attr.	100 tuples
proj_10pct	project on *signed* attr.	10% tuples

7.2.5. Aggregates

Six tests of aggregates are provided (Table 7.5.). The first is a simple scalar aggregate, computing the minimum of the *key* value. The correct query execution strategy for this aggregate consists of looking up the first value stored in the clustered index. Thus the elapsed time should be comparable to the one tuple selection with a clustered index (**sel_1_cl**). The second query is a function aggregate, which computes the minimum value of *key* partitioned into 100 partitions.

The other four queries are more complex. The **info_retrieval** query is designed to test whether systems can use bit or pointer intersection algorithms for indices to avoid a relation scan to evaluate complex predicates. The **simple_report** query is a scalar aggregate on the result of a 1:1 join of the keys of updates and hundred relations, with a 10% selection on the updates relations. The **subtotal_report** and **total_report** queries use a view that is a 1:1 join of the key of the updates and hundred relation. The SQL standard does not support the combination of subtotals and totals in the same query. One would expect this ablility in a report generator. Hence the selection/aggregate function portion of the report has been broken into two queries. Any system that has a more sophisticated report generator is allowed to rephrase and execute the two queries (**subtotal_report** and **total_report**) as a single query using the report generation facility.

Table 7.5.: Aggregate Queries

Query Name	Description	Result
scal_ag	minimum key	min value (1)
func_agg	minimum key grouped by name	100 min values
info_retrieval	select w/ complex predicate, then min(key)	1
simple_report	select avg(x) where x in (select 10%)	1
subtotal_report	10% select on view, min(a),max(a),avg(a),count(b), group by code,int	100
total_report	10% select on view,min(a),max(a),avg(a),count(b)	1

7.2.6. Updates

The update tests are designed to check both integrity and performance. To check integrity, one query that attempts to append a tuple with a duplicate key value, and a second attempts to violate referential integrity by updating a field which is a foreign key. Both queries apply to the *updates* table.

Table 7.6.: Update Tests

Test Name	Description	Tuples Updated
append_duplicate	insert duplicate key	append 1, delete 1
refer_integrity	insert invalid foreign key	insert 1, delete 1, insert 1
update_key	insert middle, last key values	insert 2, modify 2, delete 2
update_btree	1 tuple updates of btree index	insert 1, modify 1, delete 1
update_alpha	1 tuple update. of index on alpha field	insert 1, modify 1, delete 1
bulk_append	bulk insertion in mid key range	insert 1000 tuples
bulk_modify	bulk update in mid key range	modify 1000 tuples
bulk_delete	bulk delete in mid key range	delete 1000 tuples

To evaluate performance, tests that measure the overhead involved in updating each type of index are provided. Single-tuple updates and bulk updates are also provided. The single-tuple updates are grouped into three tests. Each consists of building an index, appending a tuple, modifying it, and deleting it. First the updates are performed with the *updates* relation having only one index, the clustered index on the key. Then a B-tree secondary index is built on *updates.int* and the same updates are repeated. Finally a hashed (or another available type of index) secondary index is built on *updates.code*, and the updates repeated. Each of the three tests isolates and measures the overhead involved in maintaining the corresponding type of index.

The bulk updates include an append, a modify, and a delete of 1,000 consecutive tuples, located in the middle of the *updates* relation. The update tests are sequenced so that the database is restored to its initial state when all the tests are completed. Table 7.6. contains a brief description of the update tests.

8. Multiuser Tests

There are four multiuser tests, each modelling a different workload profile:

1. An OLTP test, where all users execute a single-row update, **oltp_update**, on the same relation. This update randomly selects a single row using a clustered index and updates a nonindexed attribute. It is executed with repeatable access (Level 3 isolation)[7].
2. An Information Retrieval (IR) test, where all users execute a single-row selection query, **ir_select**, on the same relation. This query selects a single row using a clustered index. It is executed with browse access (Level 0 isolation).
3. A Mixed Workload IR Test, where one user executes a cross section of ten update and retrieval queries, and all the others execute the same IR query as in the second test.
4. A Mixed Workload OLTP Test, where one user executes a cross section of ten update and retrieval queries, and all the others execute the same OLTP query as in the first test.

The first two tests can be used to measure the throughput as a function of the number of concurrent database users; the third measures the degradation of response time for a cross section of queries caused by system load. In order to make the benchmark tractable, and to make the results meaningful with respect to the benchmark time limit and the equivalent database size metric, we have decided on strict guidelines for the execution of the multiuser tests, including the specification of the number of users according to an unambiguous formula[8]. The number of users is always equal to:

$$\frac{\text{logical database size in bytes}}{\text{4 megabytes}}$$

Hence, for the 4 megabyte database, the number of users is 1; for the 40 megabyte database, the number of users is 10. This rule for determining the multiprogramming level ensures that the system will be tested in multiuser mode. The mixed workloads contain some queries that are relegated to offline batch processing in current systems due to performance problems. Hence, given the current state of commercial software, even with a low number of users, the mixed workload tests already press the state of the art in OLTP.

---

[7] In the SQL standard there are four isolation levels called 0, 1, 2, and 3. SQL 2 level 3 is the default and is known as repeatable reads in DB2 and other systems -- it gives complete isolation from concurrent updates by other transactions. SQL 2 level 1 is called cursor stability in DB2 and avoids dirty reads but allows non-repeatable reads. SQL 2 level 0 allows dirty reads and is called browse access in NonStop SQL. If a lower degree of isolation is not supported, then the higher degree must be used. For example, cursor stability would be used in lieu of browse mode isolation for a DB2 system.

[8] Originally, we wanted to allow any numbers of that maximize throughput. However, when no think times are used, maximum throughput on systems we tested was achieved with only a few users -- sometimes just one user. So this formula was developed to prevent single-user versions of the multiuser tests. It is always instructive to plot throughputs and response times from multiuser tests as a function of the number of users.

In the cross section (Table 8.1), the retrieval queries range from simple selections of one row to a simple report query. We test the ability of the system to provide different isolation levels dynamically by executing the selection query with browse, stable, and repeatable access. All update queries in the cross section should be executed with repeatable access (Level 3 isolation). Levels of isolation required for each cross section query are shown in Table 8.1. Also included in the cross section are four large transactions that update 100 tuples. With one of these large transactions, we check for correctness in aborting a transaction.

Table 8.1.: Ten Cross Section Queries

Query Name	Description	Isolation Level
1. o_mode_tiny	overhead query	3
2. o_mode_100k	retrieves 1000 tuples (100k bytes)	2
3. select_1_ncl	1 tuple select, executed at 3 isolation levels	0,1,3
4. simple_report	select-join-aggregate	2
5. sel_100_seq	select 100 clustered tuples into temporary	3
6. sel_100_rand	select 100 random tuples into temporary	3
7. mod_100_seq_abort	update 100 sequential tuples, abort	3
8. mod_100_rand	update 100 random	3
9. unmod_100_seq	undo previous sequential update	3
10. unmod_100_rand	undo previous random update	3

At the end of the multiuser tests, correctness of the bulk sequential and random updates is checked by comparing the updated relation with the temporary tables created by Queries 5 and 6 (Table 8.1) from the cross section. The execution order for the multiuser tests is:

1. Backup updates relation, including indices, to tape or other device. This is done early on. See Appendix 3.
2. Run IR (Mix 1) test for 15 minutes.
3. Measure throughput in IR test for five minutes.
4. Replace one background IR script with the cross section script. This is the Mixed Workload IR test (Mix 3). This step is variable length.
5. Run queries to check correctness of the sequential and random bulk updates.
6. Recover updates relation from backup tape (Step 1) and log (from Steps 2, 3, 4, and 5).
7. Perform correctness checks, checkmod_100_seq and checkmod_100_rand. Remove temporary tables: sel100seq and sel100rand.
8. Run OLTP test for 15 minutes.
9. Measure throughput in IR test for five minutes.

10. Replace one background OLTP script with the cross section script. This is the Mixed Workload OLTP test (Mix 4). This step is variable length.
11. Perform correctness checks, **checkmod_100_seq** and **checkmod_100_rand**. Remove temporary tables: *sel100seq* and *sel100rand*.

The order is specified to give the system time to "warm up" before throughput is measured without allowing an arbitrary time interval for reaching steady state. The recovery test is performed after the Mixed Workload IR test in order to allow testing recovery without having too many queries in the log to rollforward.

9. Summary

In this paper, we have described a scalable and portable benchmark for use in comparing relational database systems across architectures and hardware capabilities. Portability is achieved by using industry standards where appropriate. The ANSI SQL Standard data definition language and query language are used in the specification of the attributes and queries.

There are five relations in the test database. One is a very small relation used only to measure overhead. The other four are relations with ten attributes and a scalable number of tuples. The test queries are designed so that they need not be changed when the size of the database is scaled. The benchmark consists of two modules: single-user tests and multiuser tests.

The main AS^{3}AP metric is the *equivalent database size*, which is the maximum size of the test database for which the single-user and multiuser AS^{3}AP tests can be completed in under 12 hours. The single-user tests include operational functions, such as a database load and building indices, and a complete set of relational queries and updates. The multiuser tests model mixed database workloads, including on-line transaction processing, information retrieval, and long transactions. A number of DBMS vendors have requested a copy of the benchmark, and have ported AS^{3}AP or some portion of it on their own. Many reported difficulties in running the multiuser tests. Further work would be required to make the database generator portable and to develop script models for the multiuser tests.

10. Acknowledgements

Many people have contributed to this work. First, Jim Gray deserves a special acknowledgement for following the development of AS^{3}AP from the very beginning, and contributing many ideas, comments, and criticisms. The design of the test database and the mixed workloads greatly benefited from discussions with Andrea Borr, Jim Gray and Franco Putzolu at Tandem Computers.

The Argonne National Laboratories provided partial funding for the development and implementation of the benchmark, as well as a test environment. Sohail Merchant implemented the benchmark at AT & T in Naperville, on Informix and provided useful comments on an early version of this paper.

References

[Ano85] Anon et al.,"A Measure of Transaction Processing Power," *Datamation*, 1985.

[ANSI90] American National Standards Institute, "Database Language SQL 2 (ISO Working Draft)," Jim Melton ed., ANSI X3H2-90-264. July 1990.

[BDT83] Bitton D., DeWitt D.J., and Turbyfill C., "Benchmarking Database Systems, a Systematic Approach," *Proc. Ninth International Conf. on Very Large Data Bases*, November 1983.

[BHT87] Bitton D., Hanrahan M., and C. Turbyfill, "Performance of Complex Queries in Main Memory Database Systems," *Proc. International Conf. on Third Data Engineering*, Los Angeles, February 1987.

[BMO88] Bitton D., Millman J., and Orji C., Program Documentation for DBGEN, a test database generator, University of Illinois at Chicago, 1988.

[BT85] Bitton D. and Turbyfill C., "Evaluation of a Backend Database Machine," *Proceedings of HICSS-18*, January, 1985.

[BT86] Bitton D. and Turbyfill C., "Main Memory Database Support for Office Systems: A Performance Study," *IFIP TC8 Conference on Methods and Tools for Office Systems*, Pisa, Italy, October 1986.

[BT88] Bitton D.and Turbyfill C., "A Retrospective on the Wisconsin Benchmark," *Readings in Database Systems*, edited by Michael Stonebraker, Morgan Kaufmann Publishers Inc., San Mateo, California, 1988.

[BD84] Boral H. and DeWitt D.J., "A Methodology for Database System Performance," *Proc. ACM Sigmod 1984.*

[Dew79] DeWitt D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Transactions on Computers*, Vol. C-28, N0.6.

[Kit83] Kitsuregawa, M., Tanaka H., and Moto-Oka T., "Architecture and Performance of Database Machine GRACE," *Proc. International Conference on Parallel Processing*, 1983.

[Knu73] Knuth D.E., *The Art of Computer Programming, Vol. 1*, Addison-Wesley, 1973.

[RKC87] Rubenstein W.B., Kubicar M.S., and Cattell R.G., "Benchmarking Simple Database Operations," *Proceedings ACM Sigmod 1987.*

[Ston86] Stonebraker M., *The Ingres Papers*, Addison-Wesley 1986.

[Tur87] Turbyfill C., "Comparative Benchmarking of Relational Database Systems," Ph. D. Dissertation, TR 87-871, Cornell University, September 1987.

[Tur88] Turbyfill C., "Disk Performance and Access Patterns for Mixed Database Workloads," *Database Engineering*, IEEE Computer Society Technical Committee on Database Engineering, Volume 11 No. 1, March 1988.

[YHM87] Yao S.B., Hevner A.R., Myers H.Y., "Analysis of Database System Architectures Using Benchmarks," *IEEE Transactions on Software Engineering*, 13:6, June 1987.

APPENDIX 1 - SQL SCHEMA

```
CREATE TABLE    uniques(
        key     INTEGER (4)   NOT NULL,
        int     INTEGER (4)   NOT NULL,
        signed  INTEGER (4)           ,
        float   REAL (4)      NOT NULL,
        double  DOUBLE (8)    NOT NULL,
        decim   NUMERIC(18,2) NOT NULL,
        date    DATETIME (8)  NOT NULL,
        code    CHAR (10)     NOT NULL,
        name    CHAR(20)      NOT NULL,
        address VARCHAR(20)   NOT NULL,
        PRIMARY KEY (key)             )

CREATE TABLE    hundred(
        key     INTEGER (4)   NOT NULL,
        int     INTEGER (4)   NOT NULL,
        signed  INTEGER (4)           ,
        float   REAL (4)      NOT NULL,
        double  DOUBLE (8)    NOT NULL,
        decim   NUMERIC(18,2) NOT NULL,
        date    DATETIME (8)  NOT NULL,
        code    CHAR (10)     NOT NULL,
        name    CHAR(20)      NOT NULL,
        address VARCHAR(20)   NOT NULL,
        PRIMARY KEY (key)             ,
        FOREIGN KEY (signed) REFERENCES UPDATES)

CREATE TABLE    tenpct(
        key     INTEGER (4)   NOT NULL,
        int     INTEGER (4)   NOT NULL,
        signed  INTEGER (4)           ,
        float   REAL (4)      NOT NULL,
        double  DOUBLE (8)    NOT NULL,
        decim   NUMERIC(18,2) NOT NULL,
        date    DATETIME (8)  NOT NULL,
        code    CHAR (10)     NOT NULL,
        name    CHAR(20)      NOT NULL,
        address VARCHAR(20)   NOT NULL,
        PRIMARY KEY (key,code)      )

CREATE TABLE    updates(
        key     INTEGER (4)   NOT NULL,
        int     INTEGER (4)   NOT NULL,
        signed  INTEGER (4)           ,
        float   REAL (4)      NOT NULL,
        double  DOUBLE (8)    NOT NULL,
        decim   NUMERIC(18,2) NOT NULL,
        date    DATETIME (8)  NOT NULL,
        code    CHAR (10)     NOT NULL,
        name    CHAR(20)      NOT NULL,
        address VARCHAR(20)   NOT NULL,
        PRIMARY KEY (key)             )

CREATE TABLE    tiny(
        key     INTEGER (4)   NOT NULL,
        PRIMARY KEY (key)             )
```

Comments:

In addition to a clustered index on the primary key, a number of secondary indices must be built (see Table 5.3.). The run sequence in Appendix 3 interleaves the commands to build indices with queries in the single-user tests, as a way to flush the buffers without incurring additional overhead. Schema declarations may have to be modified by using a longer *address* field and/or adding a fill field. Systems that do not support variable length strings should be penalized by carrying the maximum length of the *address* field, 80 bytes, instead of the average 20 bytes. Some systems may also not support other data types in the form we have shown in the above schema representation.

The following example shows how the schema should be modified for a system that does not support variable length strings and does not support decimal and date types as specified. In this example, the 8 byte MONEY data type is substituted for the 18 byte NUMERIC data type and the 12 byte DATE is substituted for the 8 byte DATETIME. The MONEY and DATE data types combined occupy 20 bytes, whereas the NUMERIC and DATETIME data type combined occupy 26 bytes. As the substituted data types occupy 6 bytes less than the standard data types, the fill attribute is set to 6 bytes.

```
CREATE TABLE    uniques(
        key     INTEGER4      NOT NULL,
        int     INTEGER4      NOT NULL,
        signed  INTEGER4              ,
        decim   MONEY         NOT NULL,
        date    DATE          NOT NULL,
        float   FLOAT4        NOT NULL,
        double  FLOAT8        NOT NULL,
        code    C10           NOT NULL,
        name    C20           NOT NULL,
        address TEXT(80)      NOT NULL,
        fill    C6                    )
```

APPENDIX 2 - SINGLE-USER TESTS

Operational Issues

load:

```
/* file names for flat files generated by db generator are asap .uniques, etc.  */
INSERT INTO  uniques select * from asap.uniques
INSERT INTO  hundred select * from asap.hundred
INSERT INTO  tenpct  select * from asap.tenpct
INSERT INTO  updates select * from asap.updates
INSERT INTO  tiny    select * from asap.tiny
```

if load operation does not build clustered index on key, then build clustered index on key attribute of uniques relation:

CREATE UNIQUE INDEX uniques_idx ON uniques (key) CLUSTER

build secondary B-tree index: CREATE INDEX tenpct_int_bt ON tenpct (int) BTREE

build secondary hashed index:

CREATE INDEX tenpct_code_h ON tenpct (code) HASH

similar indices are built on other relations (See Table 6.3. and Appendix 3).

table_scan:

```
select * from uniques where int = 1
```

Queries Testing Output Modes (Table 7.1)

Each Query Repeated three times: output to screen, file, and relation

o_mode_tiny:

```
select * from tiny
```

o_mode_1k:

```
select * from updates where key <= 10
```

o_mode_10k:

```
select * from hundred where key <= 100
```

o_mode_100k:

```
select * from hundred where key <= 1000
```

Selections (Table 7.2)

sel_1_cl:

```
select key, int, signed, code, double, name
from updates
where key = 1000
```

sel_1_ncl:

```
select key, int, signed, code, double, name
from updates
where code = 'BENCHMARKS'
```

sel_10pct_cl:

```
select key, int, signed, code, double, name
from uniques
where key <= 100000000
```

sel_100_cl:

```
select key, int, signed, code, double, name
from updates
where key <= 100
```

sel_100_ncl:

```
select key, int, signed, code, double, name
from updates
where int <= 100
```

sel_10pct_ncl:

```
select key, int, signed, code, double, name
from tenpct
where name = 'THE+ASAP+BENCHMARKS+'
```

variable_select:

```
select key, int, signed, code, double, name
from tenpct
where signed < :prog_var
```

variable_select: with low selectivity, index should be used:

```
Set :prog_var = -500,000,000.
```

variable_select: with high selectivity, index should not be used:

```
Set :prog_var = -250,000,000.
```

Joins (Table 7.3)

join_2_cl:

```
select uniques.signed, uniques.name,
       hundred.signed, hundred.name
from uniques, hundred
where uniques.key = hundred.key
  and uniques.key = 1000
```

join_2_n_cl:

```
select uniques.signed, uniques.name,
       hundred.signed, hundred.name
from uniques, hundred
where uniques.code = hundred.code
```

```
        and uniques.code = 'BENCHMARKS'
```

join_2:

```
      select uniques.signed, uniques.name,
             hundred.signed, hundred.name
      from uniques, hundred
      where uniques.address = hundred.address
        and uniques.address = 'SILICON VALLEY'
```

join_3_cl:

```
      select uniques.signed, uniques.date,
             hundred.signed, hundred.date,
             tenpct.signed,  tenpct.date
      from uniques, hundred, tenpct
      where uniques.key = hundred.key
        and uniques.key = tenpct.key
        and uniques.key = 1000
```

join_3_ncl:

```
      select uniques.signed, uniques.date,
             hundred.signed, hundred.date,
             tenpct.signed,  tenpct.date
      from uniques, hundred, tenpct
      where uniques.code = hundred.code
        and uniques.code = tenpct.code
        and uniques.code = 'BENCHMARKS'
```

join_4_cl:

```
      select uniques.date, hundred.date,
             tenpct.date,  updates.date
      from uniques, hundred, tenpct, updates
      where uniques.key = hundred.key
        and uniques.key = tenpct.key
        and uniques.key = updates.key
        and uniques.key = 1000
```

join_4_ncl:

```
      select uniques.date, hundred.date,
             tenpct.date,  updates.date
      from uniques, hundred, tenpct, updates
      where uniques.code = hundred.code
        and uniques.code = tenpct.code
        and uniques.code = updates.code
        and uniques.code = 'BENCHMARKS'
```

join_1_10:

```
      select uniques.key, uniques.name,
             tenpct.name, tenpct.signed
      from uniques, tenpct
      where uniques.key = tenpct.signed
```

```
            and uniques.key in(500000000, 600000000, 700000000,
                               800000000,900000000)
```

Projections (Table 7.4)

proj_100:

```
select unique address, signed from hundred
```

proj_10pct:

```
select unique signed from tenpct
```

Aggregates (Table 7.5)

scal_agg:

```
select min(key) from uniques
```

func_agg:

```
select min(key)
from hundred
group by name
```

info_retrieval:

```
select count(key)
from tenpct
where name = 'THE+ASAP+BENCHMARKS+'
  and int <= 100000000
  and signed between 1 and 99999999
  and not (float between -450000000 and 450000000)
  and double > 600000000
  and decim  < -600000000
```

simple_report:

```
select avg(updates.decim)
from updates
where updates.key in
     (select updates.key
     from updates, hundred
     where hundred.key = updates.key
       and updates.decim > 980000000)
```

create_view

```
create view
     reportview(key,signed,date,decim,name,code,int) as
          select updates.key,  updates.signed,
                 updates.date, updates.decim,
                 hundred.name, hundred.code,
hundred.int
          from updates, hundred
          where updates.key = hundred.key
```

subtotal_report

```
select avg(signed), min(signed), max(signed),
       max(date), min(date),
       count(distinct name), count(name),
       code, int
from reportview
where decim > 980000000
group by code, int
```

total_report

```
select avg(signed), min(signed), max(signed),
       max(date), min(date),
       count(distinct name), count(name),
       count(code), count(int)
from reportview
where decim > 980000000
```

Updates (Table 7.6)

append_duplicate:

/* try to append duplicate key value */

```
insert into updates values
     (6000, 0, 60000, 39997.90, 50005.00, 50005.00,
     '11/10/85', 'CONTROLLER', 'ALICE IN WONDERLAND',
     'UNIVERSITY OF ILLINOIS AT CHICAGO ')
```

remove_duplicate:

```
delete updates where key = 6000 and int = 0
```

Test of referential integrity:

/* make temp relation for restore */

```
create table integrity_temp like hundred :
insert into integrity_temp
     select *
     from hundred
     where int = 0
```

integrity_test:

```
update hundred
set signed =  -500000000
where int = 0
```

integrity_restore:

/* restore *hundred* relation in case test failed */

```
delete hundred where int = 0
insert into hundred select * from integrity_temp
```

One tuple updates, in middle and end of key range:

app_t_mid:

```
insert into updates
 values  (5005, 5005, 50005, 50005.00, 50005.00,
          50005.00,'1/1/88',  'CONTROLLER',
          'ALICE IN WONDERLAND',
          'UNIVERSITY OF ILLINOIS AT CHICAGO ' )
```

mod_t_mid:

```
update updates
set key = -5000
where key = 5005
```

del_t_mid:

```
delete updates
where key = -5000
```

app_t_end:

```
insert into updates
values  (1000000001, 50005, 50005, 50005.00, 50005.00,
          50005.00, '1/1/88', 'CONTROLLER',
         'ALICE IN WONDERLAND',
         'UNIVERSITY OF ILLINOIS AT CHICAGO ' )
```

mod_t_end:

```
update updates
set key = -1000
where key = 1000000001
```

del_t_end:

```
delete updates where key = -1000
```

/* test other indices updates */
/* test b-tree secondary index */

app_t_mid (as above)

mod_t_int:

```
update updates
set int = 50015
where key = 5005
```

del_t_mid (as above)
/* test hash index on alphanumeric field */

app_t_mid (as above)

mod_t_cod:

```
update updates
set code = 'SQL+GROUPS'
where key = 5005
```

del_t_mid (as above)

/* Bulk Updates */

bulk_save:

```
insert into saveupdates
select * from updates
where key between 5000 and 5999
```

bulk_append:

```
insert into updates
select * from saveupdates
```

bulk_modify:

```
update updates
set key = key - 100000
where key between 5000 and  5999
```

bulk_delete:

```
delete updates
where key < 0
```

APPENDIX 3 - RUN SEQUENCE FOR SINGLE-USER TESTS

This appendix lists the single-user tests in the order that they should be executed. This order is different from the logical order in which queries were listed in Appendix 2, because the operational tests and queries are interleaved in a way that prevents data from remaining in the buffer. The sequence below should be embedded in a host program, with one timing statement per test. The host program should be run in standalone mode. The embedded timing statements will provide standalone response time of precompiled queries.

Create and load database with clustered index on primary keys if provided by load utility.

Backup *updates* relation.

If index not built during load, build primary clustered index on key attribute of *updates***.**

o_mode_tiny into a relation.

If index not built during load, build primary clustered composite index on key and code attributes of *tenpcnt* relation.

o_mode_tiny into a file.

sel_1_cl

If index not built during load, build primary clustered index on key attribute of *hundred* relation.

o_mode_tiny to the screen.

Build secondary nonclustered B-tree index on int attribute of *tenpcnt*.

o_mode_100k into a file.

If index not built during load, build primary clustered index on key of *uniques* relation.

o_mode_1k into a relation.

Build secondary nonclustered B-tree index on signed attribute in *tenpcnt*.

Build secondary nonclustered hashed index on code attribute of *uniques*.

o_mode_1k into a file.

Build secondary nonclustered B-tree index on double attribute of *tenpcnt* .

Build secondary nonclustered B-tree index on decim attribute of *updates*.

o_mode_10k into a file**.**

Build secondary nonclustered B-tree index on float attribute of *tenpcnt* .

Build secondary nonclustered B-tree index on int attribute of *updates*.

Build secondary nonclustered B-tree index on decim attribute of *tenpcnt* .

Build secondary nonclustered hashed index on code attribute of *hundred*.

o_mode_10k into a relation.

Build secondary nonclustered hashed index on name attribute of *tenpcnt* .

Build secondary nonclustered hashed index on code attribute of *updates*.

o_mode_100k into a relation.

Build secondary nonclustered hashed index on code attribute of *tenpcnt* .

Build secondary nonclustered B-tree index on double attribute of *updates*.

o_mode_1k to the screen.

o_mode_10k to the screen.

o_mode_100k to the screen.

join_3_cl

sel_100_ncl

table_scan

func_agg

scal_agg

sel_100_cl

join_3_ncl

sel_10pct_ncl

simple_report

info_retrieval

create_view

subtotal_report

total_report

join_2_cl

join_2

variable_select -- with low selectivity - should use index.

variable_select -- with high selectivity - should not use index.

join_4_cl

proj_100

join_4_ncl

proj_10pct

sel_1_ncl

join_2_ncl

integrity_temp

integrity_test

integrity_restore

Remove all secondary indices from *updates*.

bulk_save

bulk_modify

append_duplicate

remove_duplicate

app_t_mid

mod_t_mid

del_t_mid

app_t_end

mod_t_end

del_t_end

Build secondary nonclustered hashed index on code attribute of *updates*.

app_t_mid

mod_t_cod

del_t_mid

Build secondary nonclustered B-tree index on int attribute of *updates*.

app_t_mid

mod_t_int

del_t_mid

bulk_append

bulk_delete

APPENDIX 4 - MULTIUSER TESTS

As described in Section 8, there are four modules of multiuser tests: OLTP, IR, Mixed OLTP, and Mixed IR. We suggest that each module be set to run with one script; within one script, for each user, a concurrent program including queries and timing statements should be forked out. No think time or presentation services are included in the measurements. The queries required for the multiuser tests are listed below

/* 1 tuple update for OLTP and Mixed OLTP tests */

oltp_update:

```
update updates
set signed = signed + 1
where key = :random_number
```

/* each user program starts with a different seed and generates a sequence of random numbers */

/* 1 tuple selection for IR and Mixed IR test */

ir_select:

```
select key, code, date, signed, name
from updates
where key = :random_number
```

Cross Section Queries for Mixed IR and Mixed OLTP tests

o_mode_tiny:

```
select * from tiny
```

o_mode_100k:

```
select * from hundred where key<=1000
```

/* this selection query should be repeated 3 times with 3 isolation levels */

sel_1_ncl:

```
select key, int, signed, code, double, name
from updates
where code = 'BENCHMARKS'
```

simple_report:

```
select avg(updates.decim)
from updates
where updates.key in
     (select updates.key
     from updates, hundred
     where hundred.key = updates.key
       and updates.decim > 980000000)
```

sel_100_seq:

```
insert into sel100seq
     select * from updates
     where updates.key between 1001 and 1100
```

/* build a temporary relation for random updates */

sel_100_rand:

```
insert into sel100rand
     select * from updates
     where updates.int between 1001 and 1100
```

/* abort update of sequential records */

mod_100_seq:

```
begin work
update updates
     set double = double + 100000000
     where key between 1001 and 1100
rollback work
```

/* updates some records randomly */

mod_100_rand:

```
update updates
set double = double + 100000000
where int between 1001 and 1100
```

/* restore (unmodify) the sequential records */

unmod_100_seq:

```
update updates
set double = double - 100000000
where key between 1001 and 1100
```

/* unmodify the random records */

unmod_100_rand

```
update updates
 set double = double - 100000000
where key between 1001 and 1100
```

At the end of the multiuser tests, the following two queries are run to check that the updates were performed correctly:

/* check correctness of the sequential updates */

checkmod_100_seq:

```
select count(*)
from updates, sel100seq
where updates.key = sel100seq.key
  and  not updates.double = sel100seq.double
```

/* check correctness of the random updates */

checkmod_100_rand:

```
select count(*) from updates, sel100rand
     where updates.int = sel100rand.int
       and not updates.double = sel100rand.double
```