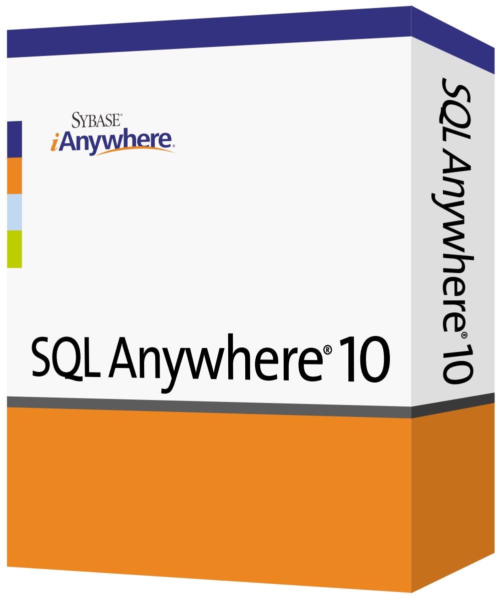
A whitepaper from iAnywhere Solutions, Inc.,
a subsidiary of Sybase Inc.



Capacity planning with SQL Anywhere

G. N. Paulley Ivan T. Bowman

19 March 2008

Abstract

We present an overview of issues to consider when designing a performance evaluation of SQL Anywhere. In particular, we outline potential pitfalls when designing a benchmark based on a specific database application. We address issues related both to the construction of a model database instance, and a model application workload, and outline an approach to construct a model of performance to determine the relative importance of various performance factors, such as database cache size and the server multiprogramming level. The paper contains several examples illustrating the potential of *negative scalability* that can occur when executing concurrent requests in an SMP environment, and how these scenarios can be generated easily by artificial contention caused by poorly-defined workloads.

Contents

List of Figures

List of Examples

List of Pitfalls

1 Executive Summary

Database vendors, along with the academic community, continue to improve the performance and scalability of commercial database management systems by testing them against well-known industry-standard benchmarks such as TPC-H [38]. However, it is difficult to determine how the results of these benchmarks apply to a particular application workload. Moreover, it is difficult to assess the scalability of a database application without thorough application-specific testing, since many performance bottlenecks are intrinsic to the application's design, rather than to any particular characteristic of the DBMS. Hence there remains a need to conduct performance evaluation and capacity planning testing on a well-defined application workload. The purpose of this technical whitepaper is to present a methodology for conducting these types of performance evaluation on a systematic basis.

In this methodology, a critical aspect is to conduct performance evaluation on a representative workload. If the workload is not representative, one of two outcomes of the performance evaluation is likely: either the actual system's performance will be worse than that predicted by the evaluation, or significant effort will be expended during the evaluation to solve performance issues that do not exist. In other words, a representative workload, coupled with solid analysis, leads to confidence in the results of the evaluation and greater productivity by reducing the risk of analyzing spurious performance problems.

Nonetheless, it is extremely tempting to develop expedient, trivial 'benchmarks' of database application performance and utilize these 'benchmarks' in comparisons of different hardware platforms, database management systems, or workload sizes, even though their results usually yield little insight to application performance behaviour. It is tempting to do so because developing a meaningful, representative workload of the system to be tested can be extremely difficult and time-consuming, for a number of reasons. They include:

- difficulty in defining a representative database instance with which to test, that:
  - contains a realistic amount of data skew and correlations found in production databases for that application; and
  - avoids artificial contention points that will cause performance problems that are unlikely or impossible to occur in a production environment;
- effort to construct a workload of database requests that representatively model the statements, their order, and timing performed by the application in a production setting;
- complications due to update portions of the workload that hamper the repeatability or concurrency of any specific experiment.

Despite these pitfalls, crafting a realistic, synthetic database workload is essential to capacity planning and performance measurement.

In addition to the workload specification, it is equally important to systematically conduct performance experiments to produce a model of application performance that can be used to guide, among other things, the hardware procurement process for a specific installation. Systematic analysis of a database application's *primary performance factors* is necessary because these performance factors—such as buffer pool size, database page size, server multiprogramming level, and the hardware platform's characteristics—may interact in different ways and may or may not be important for a given workload configuration.

In this paper, we outline several broad classes of experiment designs, and describe in detail a $2^k r$-factorial class of experimental designs that systematically analyze the changes to overall performance of k performance factors, where individual experiments are repeated r times. Linear regression analysis is then used to construct a mathematical model of application performance using these experimental outcomes.

As with workload specification, experiment design includes several pitfalls to trap the performance analyst. One is the temptation to include in the performance study as many performance factors as possible, which can lead to *multicollinearity effects* and exponentially increases the number of experiments necessary for the analysis. Another pitfall is the *correlation* of multiple performance factors, which can lead the analyst to draw erroneous conclusions about the performance data.

A well-defined synthetic workload, and a well-designed, realistic set of experiments to test database application performance, permit the construction of a model that can apportion performance increases or decreases to specific performance factors, or their combinations. This model of application performance is an important tool in the capacity planning process.

2 Introduction

Over the past several years, significant efforts have been made by both commercial database vendors and the academic community to establish well-defined database benchmarks that can be used to evaluate different relational database management systems (DBMS) in a systematic way [2–4, 16, 19, 29, 34, 37–39]. While these benchmark standardization efforts have led to significant performance improvements in commercial products, it remains difficult to draw any conclusions from industry-standard benchmark results and apply them to a real-world database application [25]. Hence there remains a need to conduct performance testing with a customer's specific database application.

One may consider undertaking a performance evaluation of a database system for several reasons. One may wish to compare different database management systems as a step in the procurement process. Or, one may wish to perform capacity planning with an existing application, either to determine what hardware requirements are needed to meet a given performance target, or to determine if a given configuration can handle a hypothetical workload. Or one may wish to determine what performance gains may be accrued by upgrading to a newer maintenance release, or major version, of the same DBMS. A unifying purpose behind these different motivations is to improve a system's efficiency, usually measured in economic terms: the cost-performance ratio [13].

Sawyer [32] outlines three fundamental questions that must be addressed at the start of any performance evaluation process, and re-examined—possibly many times—during the process:

1. What do you want to learn?
2. How much are you prepared to invest?
3. What are you prepared to give up?

All three questions are interrelated, but in practice, usually the answer to the second question dominates the other two. Conducting a detailed performance evaluation can be an exceedingly detailed, labour-intensive process that may encompass several elapsed months and significant person-months of resourcing.

One should always consider the cost of an evaluation study in addition to its advantages. Both of these are typically unknown at the beginning of the project, because it is difficult to estimate the amount of effort that will be required diagnosing issues that the evaluation uncovers, and at the same time the benefit of the evaluation will depend on how far the existing software configuration is from its optimal operating conditions [13].

An alternative to a performance evaluation study is a qualitative configuration. For example, when determining hardware requirements for a particular application, one may assume a CPU requirement based simply on the number of users, rather than any quantitative data. One such example is the IBM capacity planning guide for pSeries hardware [15]. However, rule-of-thumb capacity planning, by design, leads to *overspecification* since the overall system must have the capacity to handle peak workloads.

Pitfall 1 (Failing to develop a representative workload)
It is relatively straightforward to implement a 'benchmark' that performs a series of performance tests and generates a set of timing results. It is exceedingly difficult to develop a scalable, robust, and *representative* workload to use in a performance evaluation that provides meaningful insight. If the workload is not representative, one of two outcomes is likely: either the actual system's performance will be worse than that predicted by the evaluation, or significant effort will be expended during the evaluation to solve performance issues that will not occur in practice. □

It is also important to realize that the overall performance of a software system is most affected by the smallest *bottleneck,* though there may be several of these. In a database application, these bottlenecks may be hardware-related (CPU speed, amount of memory, characteristics of the I/O subsystem), database-system related (quality of access plans, buffer pool management, request scheduling), or application-related (design of algorithms, characteristics of individual queries, concurrency issues due to locking). Much of the effort involved in diagnosing poor performance is to isolate cause from effect. Once a bottleneck has been discovered and eliminated, diagnosis and experimentation continues with the next bottleneck until either the project's investment limit is exceeded, or the goals of the performance evaluation are met.

Pitfall 2 (The importance of application scalability)
How well a database application scales often depends as much on characteristics of the application as it does on the scalability of the database server. During a performance evaluation, application changes to improve its scalability are the norm, rather than the exception, and should be planned for. □

Pitfall 3 (Underestimating performance dependencies)
Database applications are tuned, either explicitly or implicitly, to execute well with a specific release of a database system. When migrating to a new database server release, or a different DBMS altogether, one will undoubtedly discover performance regressions that may be difficult to analyze. In our experience, these issues usually impose performance bottlenecks that must be identified and solved in order for the performance evaluation to proceed. These issues often benefit from technical support or on-site consulting assistance. □

In this technical whitepaper, we outline some of the methodologies and pitfalls in conducting a performance evaluation of a database application utilizing SQL Anywhere. We assume that the performance evaluation involves an existing SQL Anywhere database application.

Throughput	Amount of useful work completed per time unit with a given workload
Relative throughput	Ratio of elapsed time to process given workload on a system S_1 with another system S_2
Capacity	Maximum amount of useful work that can be performed per unit time with given workload
Response time	The time interval between the submission of an interactive request and when its result is returned to the client, usually expressed as a distribution
Turnaround time	In a batch environment, the time between job submission and completion
Price/performance	System cost per workload unit. A workload unit is typically a metric such as transactions per second, though other measures, such as average response time, can also be used

Table 1: Performance measures of application-oriented system effectiveness, adapted from Svobodova [36] and Ferrari et al. [13].

The outline of the rest of the whitepaper is as follows. In Section 3 we outline an approach to performance evaluation through the setting of specific, measurable goals. In Section 4, we describe a methodology for constructing a representative workload—including both the database instance and the application workload. We also discuss contention phenomena in detail in Section 4.3.4. Finally, in Section 5 we describe the primary factors to consider in the setup of experiments to measure system performance given a certain workload. We conclude the paper in Section 6.

3 Goals

While most think of benchmarks as simply a measure of performance (see Table 1), performing a performance evaluation may lead to other discoveries that are not simple measures of raw speed. These include aspects of system functionality, usability factors, ease of operation, ease of development, and so on [32]. However, in this whitepaper we address only performance issues and not these other aspects.

It is good practice to decide in some detail what the goals of the performance evaluation are before proceeding with the project, since the goals may dictate changes to one's overall approach [12–14, 32]. For example, if the goal is to conduct capacity planning tests to determine the characteristics of the hard-

Interarrival time	In an interactive application, the time required by a user to generate a new request; also known as 'user think time'
Request intensity	Ratio between the mean response time and mean think time
Priority	Relative priority of a request, as assigned by the user or application
Working set size	Number of pages in the database buffer pool that remain resident during a trial
Locality of reference	Measure of the proportion of the data that is accessed by the requests in the entire workload
User intensity	Ratio of processing time per request to user think time
Number of users	Number of concurrent connections
Number of concurrent requests	Number of concurrent requests either being executed or waiting in queues
Request mix	The distribution of different classes of requests contained in the workload

Table 2: Workload characterization variables, adapted from Svobodova [36] and Ferrari et al. [13].

ware necessary to support a given workload, or determine if the workload's response time is within an acceptable range, then it is important that the benchmark's workload mirror the real world as closely as possible; variables such as transaction interarrival time (see Table 2) will be critical to the accuracy of the test results. With capacity planning, the typical consequence of inaccuracy is the need to overspecify the system's hardware requirements, which is the underlying methodology to simplistic 'cookbook' approaches for system requirements [17].

On the other hand, if the goal of the performance evaluation is to assess the performance of different database systems with the same workload, or assess the capabilities of different hardware systems with the same workload and system configuration, then the precise accuracy of the workload's characteristics may be subject to other tradeoffs. Collectively these goals are termed *load testing*; other potential goals include *stress testing* and *spike testing*. In a stress test, the workload is constructed to mimic a worst case scenario. With interactive applications this is typically done by artificially setting the transaction interarrival time to zero, or near zero. In a spike test, the workload is deliberately skewed to periodically generate a series of database requests several times larger than average [27].

4 Workload specification

It is convenient to think of a database application workload in three parts: the hardware configuration upon which the performance test will be done; the database instance used for the performance evaluation; and the application workload per se, which consists of application logic and SQL requests that execute over the database instance. In a performance evaluation, it is important that the hardware, network, and operating system mirror the actual system as closely as possible. However, in a test involving hundreds of simulated users it may not be possible to reproduce the actual hardware environment in the lab. Similarly, both the database instance and the application workload are *models* of the actual system. There are no absolute or unique criteria for building and evaluating workload models, but the accuracy of the model is an essential characteristic for the credibility of the results of the performance evaluation [13, pp. 44].

While the workload model may be an actual instance of a production system, there are several motivations for using a model instead of employing the production instance directly in an evaluation. The most important of these motivations [13] include:

1. *reproducibility.* Any performance evaluation must be reproducible, in that an evaluation may have to be performed multiple times for the sake of comparison with different system configurations.

2. *scalability.* If capacity planning is a goal of the performance evaluation, the workload must be able to scale so that the performance of different workload sizes can be analyzed.

3. *reduction.* The duration of the performance evaluation should be such that it can be conveniently run multiple times, often in significantly less time than the 'real' workload.

4. *consistency.* To maximize the representativeness of the evaluation, one must ensure that, for example, the database instance is consistent with the model application workload.

5. *security.* It is often problematic to use actual customer data in a performance evaluation since the data may be subject to disclosure during the test.

The implementation of a workload model can be classified into three broad categories: *real, synthetic,* and *artificial.*

Real workloads are actual instances of production databases, tested using production application software and, if the applications are interactive, with actual users. Real workloads, obviously, have the greatest measure of representativeness to actual systems. However, real workloads (with *real* users) are not reproducible, nor do they scale easily.

Synthetic workloads consist of a mix of real workload components along with synthetic scripts, utilities, and other specially-constructed components. Most performance evaluations utilize workloads of this type. The typical synthetic workload uses a *sample* of the real workload, usually made necessary since the actual workload is often extremely large. However, it is often assumed that the longer the observation period (sample) for the observed workload, the better the results [11].

Artificial workloads do not make use of any real components; in the main, artificial workloads are *simulations* (cf. reference [24]) that utilize analytical models of various hardware and software components.

Boral and DeWitt [4] argue that the three most significant factors that affect database system performance are:

1. the size of the buffer pool's *working set*;
2. the *multiprogramming level* of the database server; and
3. the workload *mix*.

In what follows, we look at each of these performance factors in turn.

4.1 Database specification and the working set

In general there are two categories of database instances used in performance evaluation: *synthetic* and *real*. Synthetic databases are used in all industry-standard database system benchmarks (TPC-H, TPC-C, 007, TPC-W, AS3AP, and so on). Because synthetic databases are generated via software, they are scalable, secure, reproducible, and can be constructed to exhibit various degrees of data skew, although most industry-standard benchmark databases utilize uniformly distributed data[1]. This is done not only because it is the simplest approach, but also because coupling a parameterized set of query templates to a uniformly-distributed database instance maximizes the database buffer pool's working set, and that working set will scale proportionally to different database sizes. Moreover, uniformity matches the underlying assumptions of most, if not all, query optimizers, and hence implicitly leads to more accurate access plans [6]. To put it another way, uniformly-distributed data diminishes the

1 Note that only the proposed TPC-DS benchmark [29] specifies any degree of data skew. In most other industry-standard benchmarks, database data is generated with a uniform distribution.

need for sophisticated estimation algorithms that, today, are a requirement of industrial-strength query optimizers.

However, while it is feasible to generate data with varying degrees of skew, it is very difficult to construct a *representative* synthetic database that mimics a real database instance. This is because it is hard to model the *data correlations* that all real-world databases exhibit[2].

Because constructing synthetic databases can take a significant degree of effort, often a representative production system is used as the test workload database. While such a database is representative of *one* production instance, using such a database in a benchmarking scenario also has significant drawbacks.

Firstly, there may be security concerns with the customer data. Occasionally, sensitive data can be obfuscated or eliminated from the database outright, but doing so may affect the representativeness of the database, and hence put the performance evaluation results into question.

Secondly, the application workload is often highly dependent upon the contents of the database. For example, most databases contain various date, time, and timestamp columns. Consequently, it is essential that queries in the workload that compare these columns to specific literal constants (or host variables, or stored procedure parameters) utilize date and time values that match the contents of the database.

Thirdly, such a database is difficult to scale, for at least three reasons. It is difficult to reduce or augment the real-world data in the database yet still mimic the existing distributions and correlations amongst the various schema attributes. It is also difficult to avoid violating any referential integrity constraints that may exist. Most importantly, it is difficult to scale the database in such a way that the working set of database pages that will be required by the application workload during the evaluation is realistic. Moreover, during experiments it is important to set the size of the database server's buffer pool (cache) appropriately with respect to the (scaled) working set[3].

Pitfall 4 (Instance- or time-specific values)
SQL requests that refer to `CURRENT DATE` or `CURRENT TIME`, or other system-specific or time-dependent system variables, almost certainly require modification before including them in the workload. □

Pitfall 5 (The danger of using production databases)
When designing a performance evaluation for capacity planning purposes, it may be tempting to use a real-world production database instance for the test,

2 For an example of how difficult it is to do this, even with a simplistic approach, see reference [9, Chapter 7].

3 Recall that SQL Anywhere supports dynamic cache sizing, which alters the size of the server's buffer pool automatically upon demand, providing another workload parameter that must be controlled during measurement experiments.

'growing' the database by artificial means to represent larger installations. However, this approach usually fails to alter the working set of the application significantly, and thus is not representative of the performance results that would be seen in practice. At the same time, with larger numbers of concurrent requests, it is likely that artificial contention will be introduced amongst them, resulting in reduced throughput (see Section 4.3.4). □

4.2 Setting the multiprogramming level

With SQL Anywhere, the server multiprogramming level is simply the maximum number of concurrent tasks, which corresponds to the number of threads (or fibers, depending on platform) created within the database server process. This number is controlled by the `-gn` command line switch. For a network server, the default is 20. Setting `-gn` to a higher value permits a larger number of tasks to execute concurrently. Ordinarily, a request to the server from an application or stored procedure is an SQL statement, such as `FETCH`, `CALL`, `OPEN CURSOR`, or `SELECT`, that is executed as one task within the server. With intra-query parallelism supported in SQL Anywhere Version 10, however, a single request can be processed by multiple tasks simultaneously.

Setting a higher multiprogramming level is done to increase throughput, but there are tradeoffs. With some workloads, the extra degree of concurrency may cause a *reduction* in throughput because the database server may spend a greater amount of resources context-switching between the tasks, a phenomena known as *thrashing.*

A higher level of multiprogramming may also adversely affect the response time of individual requests, for a variety of reasons. Increasing the multiprogramming level may lead to greater contention simply because the probability of concurrently executing, but conflicting, requests is increased. Increased values of `-gn` will also reduce the amount of available memory to execute the request. Locally, the amount of memory available for the execution of each database request is governed by two memory-consumption thresholds:

- a *hard* limit, which, when exceeded, causes the request to fail with an error. This hard limit[4] is computed as

$$\frac{3}{4} \times \frac{\text{maximum cache size} - \text{adjustment value}}{\max(\text{number of active tasks, } 1)} \tag{1}$$

- a *soft* limit, which the server tries not to exceed through strategies such as requesting execution operators to free memory during query execution,

4 Incorrect information about the soft and hard memory governor limits appears in the SQL Anywhere 9.x and 10.x documentation.

or by dynamically switching to a different physical operator that requires fewer memory resources. This limit is

$$\frac{\text{maximum cache size} - \text{adjustment value}}{\max(\text{ multiprogramming level, 2 })} \tag{2}$$

In the two formulas above, *maximum cache size* refers to the maximum size that the database buffer pool could reach[5], which can be set explicitly using the `-ch` server command line switch. The *adjustment value* is the sum of buffer pool pages used for internal representations of schema and connection objects, incremented by a 100-page safety margin (10 database pages on the Windows CE platform). The purpose of the adjustment value to eliminate from consideration those buffer pool pages that must remain resident in memory.

Globally, increasing the server multiprogramming level will slightly reduce the effective size of the database cache, because each task within the database server is allocated a specific amount of virtual address space to be used as stack space. On Microsoft Windows systems, each stack consumes one megabyte of address space, which correspondingly reduces the amount of available virtual memory for the rest of the database server process.

On 32-bit Microsoft Windows XP systems[6] without Advanced Windowing Extensions (AWE) support, the maximum available database cache that can be used is approximately 1.6 to 1.8 GB. Increasing the number of concurrent requests using `-gn` reduces this amount by slightly more than one megabyte per additional task. Note that the address space requirements are independent of the amount of physical RAM available in the system. However, for best performance, one should ensure that the amount of physical RAM is larger than the cumulative size of memory requirements for the database server process, other concurrent applications, and all OS processes, in addition to the database server cache size. Otherwise, the operating system will be forced to swap elements of the database server's memory to disk, which will significantly reduce performance.

On 32-bit Linux platforms, the maximum size of the database cache that can be allocated depends on the specific Linux distribution (kernel configuration) that is being used. The most common case is to have 2 GB of address space available for the database server process, meaning that the maximum database cache size is approximately the same as in most Microsoft Windows environments. On most other 32-bit UNIX systems, however, allocated virtual address

5 The hard limit can exceed the current buffer pool size if automatic resizing of the cache is disabled through the `-ca 0` command line switch.

6 On 32-bit Windows Advanced Server platforms, the maximum available address space available for the database server process (without Advanced Windowing Extensions) is approximately 1 GB larger (2.6 GB in total).

space is immediately backed by physical RAM or swap space. Hence, the effects of a larger `-gn` value are more pronounced. A higher multiprogramming level reduces the amount of virtual address space available for the buffer pool. In addition, because the operating system immediately allocates physical memory to support that address space—even if that memory is not currently in use—the effective amount of memory is reduced for the entire system.

For all practical purposes, these limits do not apply to 64-bit Microsoft Windows, Linux, or UNIX platforms. However, one must still ensure that enough physical RAM is available to support the virtual memory requirements of the system—otherwise, swapping will occur and performance will suffer. To some extent the characteristics of the workload dictate the effects of the stack resources required and, consequently, the amount of physical RAM required to support the database server process.

4.3 Application workload specification

4.3.1 Approaches to implementing the application workload model

Workload model implementations can be built in a variety of ways, and which techniques are used are often influenced by the goals of the performance evaluation [13, 27]. A workload implementation may be constructed from a subset of the production system, including the use of actual system components. Industry-standard benchmarks, such as the TPC-H decision-support benchmark [38], utilize a completely synthetic approach involving a specific set of parameterized *query templates*. In the case of TPC-H, each `SELECT` and `UPDATE` statement is executed with a set of host variables, whose values are assigned according to a specific distribution defined by the benchmark. More commonly, a workload model for a database application is constructed using *traces*, that is, a detailed log of database requests that represents application activity.

With SQL Anywhere, acquiring an application trace is straightforward with the use of application profiling (or request-level logging, in older SQL Anywhere releases)—each statement issued by the application to the database server is logged in chronological order. Short of testing actual production programs, such a trace offers the greatest representativeness, as it contains the precise sequence of SQL requests that have been executed by the database server[7].

7 Note that while a request-level log does capture an application's server activity, it does not capture the set of SQL requests that have been invoked within stored procedures or functions as a result of those application requests. The application profiling feature in Version 10 of SQL Anywhere does, however, offer this functionality. Neither application profiling nor request-level log output is in a form suitable for direct use in a performance evaluation, and will require some degree of post-processing for use in a script.

PITFALL 6 (USING APPLICATION TRACES AS A SYNTHETIC WORKLOAD) Utilizing SQL traces of actual application activity does have several disadvantages:

1. To be representative, the trace must be captured on a real production system, possibly with significant inconvenience.

2. Any trace is directly 'tied' to a *specific* database instance, and may be difficult or impossible to use with other databases, affecting both the representativeness and reproducibility of the workload.

3. It is difficult to partition the trace, using sampling or other techniques, without affecting its representativeness.

4. Since a trace is the log of specific connections, it may be difficult, if not impossible, to simulate a larger number of connections in the workload using the identical trace without introducing artificial contention (see below).

5. In a multi-user test, using a single trace to represent the activity of multiple users is unlikely to affect the working set of the database buffer pool in the same way as in a production system—hence the performance evaluation will almost certainly yield unrepresentative results.

6. A trace will include a specific distribution of business transactions executed by a single connection, or set of connections. In a test involving a greater number of users, this sample distribution may not hold—typical distributions of business transactions, or web server transactions, have been studied in the literature and have been shown to follow a Zipf distribution [30] (see Section 4.3.3 below). However, it is difficult to both analyze and sample occurrences of business transactions from simply a chronological log of SQL requests.

7. Updates may pose a particular problem when using a trace to generate a workload. Requests within a trace use parameter values that depend on previous updates; these requests may fail if the database is not in the same state that it was when the trace was obtained. To some extent, updates can be accommodated by making a backup copy of the database immediately before obtaining the trace, then restoring the backup before replaying the trace. If all requests in the trace are executed in their original order, the database will be in the same state as it was when the request was originally submitted. If requests can be submitted in a different order during replay, then special handling may be needed to handle the effects of updates.

□

A *completely* synthetic implementation of a workload model—such as those used by industry-standard benchmarks like the TPC-H benchmark—overcomes several of the above disadvantages of using traces. However, completely synthetic workloads usually take *considerably greater effort to construct.* The most difficult part of constructing a completely synthetic workload is in achieving an acceptable degree of representativeness to the real-world system. This can be accomplished through detailed statistical analysis of the application to determine the set of *workload parameters* that must be modeled, and subsequently constructing mixes of requests that represent the important components of the application to be tested [11, 13]. The most significant pitfall in this process is to incorrectly model *correlations* amongst the various workload parameters [11]. While the effort to perform this analysis is considerable, the advantages of a completely synthetic workload implementation are:

1. workload parameters can be changed at will, independently if desired, to investigate the influence of each [11];
2. completely synthetic workloads typically offer better flexibility, reproducibility, and control [13];
3. synthetic workloads can be used with a synthetically populated database;
4. one is able to repeat performance experiments under statistically similar conditions that are nevertheless not identical [11].

Whether using SQL traces or synthetic queries to model a database application workload, the key requirements are, firstly, to ensure that the workload size is both large enough to be realistic, and provides enough stress on the particular system component being analyzed. This provides enough opportunity to measure the workload's performance once a steady-state has been reached during the evaluation. Secondly, one must avoid the introduction of artificial bottlenecks that do not, or will not, exist in the real system, as these can skew the results of the evaluation.

4.3.2 Constructing a representative workload

In any performance evaluation, a workload must be carefully crafted to mimic the behaviour of the application in a real, production environment [12, pp. 228]. Constructing a representative workload leads to greater confidence in the evaluation's results. A representative workload also helps to ensure that one does not introduce artificial bottlenecks that do not, or will not, actually exist in the real world. The most restrictive bottleneck in the entire computer system affects overall system performance to the greatest degree, and, as mentioned previously, the bottleneck can be caused by hardware constraints (memory capacity or performance, CPU speed), network bandwidth, application-caused issues such as inefficient processing techniques, or lock contention on rows or tables (see Example 1 below), or by contention on objects within the database server itself (server data structures, buffer pool pages, TCP/IP buffers, I/O synchronization primitives, and so on).

Example 1 (Artificial contention caused by application design)
Consider a database design in which several of the most important business objects are identified using surrogate keys. The application is designed to use a special table in the database, which we simply call the `surrogate` table, where each row in the table represents a type of business object. Contained within each row of the `surrogate` table is the value of the next key to be used when inserting a new business object of that type—we assume that value is stored in the column `next_key`. Within the application, the logic for inserting a new client into the database is roughly as follows:

```
UPDATE surrogate SET @x = next_key, next_key = next_key + 1
       WHERE object-type = 'client';
INSERT INTO client VALUES(@x, ...);
COMMIT;
```

This code fragment will cause the serialization of every insertion of new `client` rows, as only one connection at a time will be able to acquire a write lock on the 'client' row of the `surrogate` table. It may be the case that this logic will perform adequately with a small number of connections, but this approach is not scalable. With significant numbers of users the formation of *convoys* is inevitable (see below). □

In addition to scalability issues directly caused by the construction of the application, it is also possible to introduce artificial performance contention within the database server with a workload that does not reflect the real-world production system. One way to do this is with an unrealistic *workload mix,* where the distribution of the SQL requests in the workload does not represent the anticipated mix at that specific scale. In our experience, a typical data processing workload consists of 85% queries, 8% inserts, 5% updates, and 2% deletes. Statistical sampling techniques [11, 13] can be employed to ensure that the workload mix is appropriate at different levels of scale so that the results of the performance evaluation are credible.

Pitfall 7 (Evaluating only read-only workloads)
Constructing a performance evaluation workload using only queries usually provides merely a best-case result of overall performance. Application-caused contention will rarely occur with such workloads. The results of this sort of evaluation should be given little weight with any capacity planning assessment. □

Another way of introducing artificial contention into an interactive workload is by using an unrealistic distribution of user think time, also known as the transaction interarrival rate [32]. For synthetic workloads, common distributions of user think time are uniform, random, and negative exponential [32]. Uniform distributions are, by their definition, unrealistic; they fail to accurately represent typical user interactions. Negative exponential distributions, such as Zipf distributions (see Equation 4 in Section 4.3.3 below), offer skewed distributions that more accurately describe typical distributions of user think time. Negative exponential functions are a much better choice to model the transaction interarrival rate in performance analysis. An example of a negative exponential transaction interarrival rate appears as part of the SPEC JAppServer 2004 benchmark [34]. A trace can be used to model the transaction interarrival time, but it may be inappropriate to use the request times of a single trace to characterize a workload of hundreds of concurrent connections.

4.3.3 Zipf distributions

Zipf's Law (cf. reference [30]) describes a family of statistical distributions of the form

$$r^{\alpha} x_r = K, \alpha > 0 \tag{3}$$

sometimes rewritten as

$$x_r = \beta r^{-\alpha}, \alpha > 0 \tag{4}$$

where x_r is a specific x-value in an ordered set of N x-values $x_1 \geq x_2 \geq \ldots \geq x_N$ and r is the *statistical rank* of x_r in this order. Often the set of x-values are used to represent frequencies; the values K, α, and β are arbitrary constants. Using equation (4), varying α and/or β yields a series of rank-frequency relationships that can be graphed as rectangular hyperbolas (see Figure 1). Note that the greater the value of α, the greater the degree of skew. If α is 0, then the rank-frequency distribution is uniform.

In a database context, what Zipf's law means is that any particular column's distribution is likely to be skewed such that a few values will occur with high frequency, while at the other end there will be many values that are unique, or nearly so. In practice, most table columns have skewed distributions—with the exception, of course, of primary keys.

4.3.4 Convoy formation

SQL Anywhere 9 can use multiple processors to handle requests, although each request is handled by only one[8] processor. It is natural to assume that, if you have N processors, you can handle N requests in the time that a single request

8 SQL Anywhere 10 supports intra-query parallelism, making scalability analysis more complex since a single request can be computed using more than one processor simultaneously.

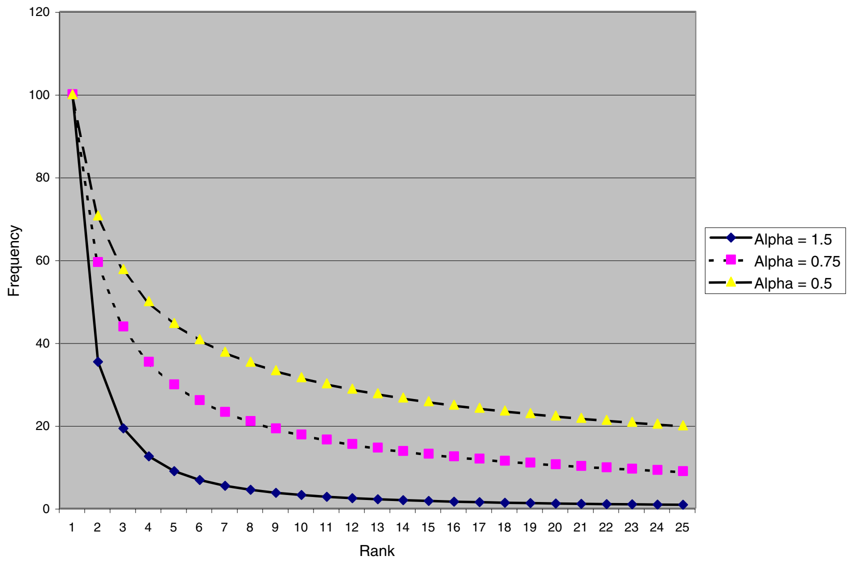


Figure 1: Three example Zipf distributions, all with $\beta = 100$.

can be executed. This is the assumption of perfect scalability. While some workloads can come close to this ideal, scalability requires careful application design.

The *raison d'etre* of a database system is to provide concurrent access to the *same* information, with guarantees of transactional acidity. Every database system implementation, commercial or research prototype, contains structural and algorithmic points of contention that the vendor has traded-off for other specific advantages. These tradeoffs typically differentiate commercial DBMS in the marketplace. Often these tradeoffs involve issues regarding the precise semantics of transactions at different isolation levels.

As a concrete example, Sybase IQ does not implement row-level locking, but instead offers a simplified concurrency control scheme known as *table level versioning,* or TLV. TLV eliminates the need for read transactions to lock rows to get a consistent view of the database, but at the (considerable) expense of permitting only one writer transaction at a time per base table. As another example, some of the physical access methods supported by IMS/TM, IBM's hierarchical database system, only support page-level locking. This tradeoff permits the IMS lock manager and recovery manager to be considerably more efficient, but can severely degrade the performance of insert transactions due to page-level lock contention. In summary, every DBMS has its limits, and every DBMS implementation contains its own bottlenecks.

The SQL Anywhere database server contains internal data structures which are shared by multiple connections. To ensure that each connection sees a logically consistent version of the data structure, *mutual exclusion* (mutex) operations are used. Mutex operations are similar to locks on rows, preventing a reader from observing an inconsistent version of a data structure while a writer is modifying the structure. Although it is not intuitively obvious, these mutex operations must be used even for read-only queries that do not modify the database. Mutex operations are needed because the engine's internal data structures, such as the cache manager, are modified even by read-only queries. For example, pages are added to and removed from the cache during read-only queries. Mutex-protected structures within the engine typically use a fast check that passes quickly if there is no possible contention, then a relatively expensive algorithm if there are other requests waiting for the same shared object. Mutex implementations, and their relative costs, vary from operating system to operating system, and this difference can be significant in a performance evaluation.

Mutex operations are similar to row locks used by transactions when modifying table rows. However, the duration of the mutual exclusion is very short, much shorter than the duration of row locks. For example, when looking for a page in the cache manager, a connection acquires a mutex object, checks a few pointer values, and releases the lock. A connection will only be affected by the mutex operation if it is attempting to access the same page as another connection. In that case, the second connection is forced to wait for a short period.

In the best case, mutex operations do not greatly affect the scalability of an application workload. If connections do not frequently need to access the same data structure objects, they will rarely need to wait for a mutex operation to complete. However, for some workloads there may be one or more *hot spots* within the engine data structures, and these hot spots may vary between software releases as performance improvements are made. For example, there may be a single table that is referenced many times by all connections. In this case, the mutex-protection mechanisms associated with engine data structures for this table are likely to be held by one connection when another connection tries to access them. This will lead to higher execution times as the connections spend time waiting for mutex operations to complete, rather than executing the query per se.

In some situations, a convoy can form. Consider a case where many connections are all executing the same join between a large table S and a small table R. Each connection fetches rows from S and searches an index on R to find matching rows. The engine has mutex operations protecting access to the pages of the index on R. Because all connections are accessing the same page, it is likely that connections will need to access the index page at the same time, leading to some connections having to wait. However, the amount of work that each connection does may be small relative to the cost of the mutex operation (which has costs associated with checking, waiting, and resuming). Thus, a convoy forms: all connections but one are waiting for a shared resource. The one active connection does some small amount of work and releases the shared resource, but in a very short period of time finds itself waiting for the resource again. In this way, the entire execution becomes serialized on the single shared object.

PITFALL 8 (THE MYTH OF PERFECT SCALABILITY)
It is commonly believed that simply improving the hardware platform by a factor of x will correspondingly improve application performance—again, this is another manifestation of the myth of perfect scalability. It is helpful to remember, in this context, that *all processors wait at the same speed.* □

The following two points of contention commonly cause the serialization of requests in SQL Anywhere 9 servers due to application design:

- *Cache chains.* SQL Anywhere maintains an internal hash table with a key of page identifiers and a value of the page in memory. Prior to version 9.0, mutex protection is used for each bucket of the hash table. The hash table is sized so that each bucket is expected to have a length of one, so typically two requests will only conflict if they are referencing the same page. This is a source of contention for 'hot pages' such as index root pages. In version 9.0 and above, read-only requests typically do not acquire a mutex operation on a cache chain, and this source of contention is substantially reduced.

- *Row references.* When accessing rows from a table, a reader process needs to ensure that it is referencing a consistent image of the row. Mutex protection is used to ensure consistent access, so that the values in the row read by the database server into memory are consistent with a single point in time, even if the row spans multiple database pages.

If multiple requests access a single point of contention, for example a hot page or the same row, then requests may serialize into a convoy on the point of contention. In this situation, all requests but one are waiting for access to the shared object. One request does some work with the shared object (usually a fairly trivial amount, such as following a pointer), and then releases the mutex object, continuing to perform work for the client request. The work of choosing another active task to execute, and switching to it, is typically very high compared to the work performed in the critical section. By the time another waiting task has been activated, the first request may again need to access the same shared object. In this way, execution is serialized, and performance is worse than executing all of the requests serially, resulting in *negative scalability.*

Example 2 (Positive scalability)
As an example[9] consider a database instance with the following schema:

```
CREATE TABLE T1( x INT PRIMARY KEY );
CREATE TABLE T2( x INT PRIMARY KEY );
INSERT INTO T1
       SELECT R1.row_num-1 + 255*(R2.row_num-1) AS x
       FROM rowgenerator R1, rowgenerator R2
       WHERE x < 1000
       ORDER BY x;
INSERT INTO T2 SELECT * FROM T1;
COMMIT;
```

and queries Q_1 and Q_2, respectively, as follows:

```
SELECT COUNT(*)
FROM T1 AS A, T1 AS B, T1 AS C
WHERE C.x < 20

SELECT COUNT(*)
FROM T2 AS A, T2 AS B, T2 AS C
WHERE C.x < 20
```

Note that queries Q_1 and Q_2 do not access the same rows. If these two queries execute concurrently, they will not generate contention for rows, data pages, or index pages. When run individually, the average request time is 31.5s. If Q_1 and Q_2 are run concurrently, the running time averages 42.3s. This is slower than the time with only one request because more CPU time is spent coordinating activities, but it is still substantially better than the time required to run the two queries sequentially, which would be $31.5 + 31.5 = 63$s. □

Example 3 (Convoy formation with join queries)
On the other hand, some queries do generate contention for shared objects. Consider the following queries, which we call Q_3 and Q_4:

```
SELECT COUNT(*)
FROM (T1 AS A, T1 AS B, T1 AS C) LEFT OUTER JOIN rowgenerator R
     ON R.row_num = A.x + B.x
WHERE C.x < 5


SELECT COUNT(*)
FROM (T2 AS A, T2 AS B, T2 AS C) LEFT OUTER JOIN rowgenerator R
     ON R.row_num = A.x + B.x
WHERE C.x < 5
```

9 In this example, performance results were obtained with SQL Anywhere 8.0.2(4294) on a dual processor Pentium XEON 2.2GHz with data fully cached; queries were timed with the FETCHTST utility; absolute times will vary from system to system.

When run individually, queries Q_3 and Q_4 take 33.2 seconds in total. However, running both concurrently gives an average running time of 74.2s for *each* query. This is significantly slower than running each query separately in sequence. The problem is that both of these queries are accessing the `RowGenerator` table 5 million times. Each query accesses the same index pages, the same table pages, and the same rows in the same order. This is very likely to lead to a convoy situation as described previously. A characteristic of a convoy is that running time is slower than a sequential execution of the queries—in other words, *negative scalability.* □

The amount of contention generated by a workload can vary depending on the amount of data cached. When the cache is mostly empty (at start up), connections spend more time waiting for I/O requests than for mutex objects, and convoys are less likely to occur. Also, the query execution plan selected in this case may be different, avoiding a plan that generates excessive contention.

Pitfall 9 (Causing convoy formation with hot rows or pages)
While many applications will not be affected by serialization on row mutex operations, there are two particular situations that can still cause convoys to form. First, multiple connections may reference the same small set of rows multiple times. This can occur if there is a lookup table, for example, that is referenced by each connection with the same key. In this case, all connections may serialize on the mutex object for these hot rows. Second, consider a scenario where multiple connections make repeated scans of a larger number of rows, always accessing the rows in the same order—for example, with a sequential scan, or through the identical index. With this type of access, there may not be any hot rows that are accessed inordinately frequently, but by using the same order to reference the rows, the probability of contention is greatly increased. A convoy may form on one row and then be continued across multiple rows in the scan order. □

In order to achieve positive scalability, applications must be designed to avoid generating excessive contention within SQL Anywhere data structures. In particular, applications should avoid referencing a small subset of rows on many connections (the hot row problem). This may be avoidable using caching on the client. In some cases, it is reasonable to copy these hot rows into a global temporary table which gives each connection a separate copy of the rows.

The worst possible situation for an SMP server occurs when many concurrent requests are likely to access the same shared objects at about the same time. In many cases, applications will access rows in different orders, and each connection is not likely to fetch the same row multiple times. In this case, it is not likely for serialization to form on individual rows. When it does form, it usually clears quickly without long-term degradation of performance. However, if the same query is executed concurrently by multiple connections using the same host variable bindings, convoys are likely.

Convoy formation is highly relevant to the creation of synthetic database workloads. Naïve construction of an application workload, particularly using multiple copies of the same application trace, constitutes the greatest danger to the credibility of a performance evaluation study because of the amount of potential contention that may be introduced.

4.3.5 Client configuration

Our focus thus far has centered on the performance characteristics of the SQL Anywhere database *server*. In a client-server environment, however, the configuration of the *client(s)* can have a significant effect on the application's overall performance.

Application performance is almost always measured at the client, since that is the point that a user will interact with the system, and the client program is the easiest place to measure performance aspects such as response time. Application programs read and write data to the database as is necessary for the application's operation—the issue with performance analysis is precisely *how* these operations take place, their latency, and their affect on system throughput.

Latency refers to the time delay between when one machine sends a packet of data and the second machine receives it. For example, if the second machine receives a packet 12 MS after the first machine sent it, the latency is 12 MS. *Network throughput,* on the other hand, refers to the amount of data that can be transferred in a given time. For example, if it takes four seconds to send 1 MB of data between two machines, the network throughput is 250 KB/s. On a LAN, latency is typically slightly less than 1 ms, and throughput is typically more than 1 MB/s. On a WAN, latency is often significantly higher (perhaps 5 MS to 500 MS), and the throughput is typically lower (perhaps 4 KB/s to 200 KB/s).

Pitfall 10 (Network performance can dominate response time) Network throughput is *considerably* slower than disk transfer times on modern server hardware. Network traffic, therefore, can be a performance killer, to the point that network transfer time may dominate the performance characteristics of the entire application—and performance tuning of the database server will yield little, if any improvement. For high network latency, a worthwhile goal is to reduce the number of requests sent to the server, which will reduce the number of network round trips. For poor throughput, the amount of data transferred between the client and server can be reduced, if possible, to improve performance [26]. □

For a network with relatively high latency[10], one can use the following strategies to reduce the number of client/server requests:

- minimize the number of client requests sent to the server by combining SQL requests together;
- refrain from performing joins within the application itself by using multiple queries;
- use bound columns, rather than `Get Data`, to retrieve attribute values;
- utilize network prefetching to reduce the number of `FETCH` requests sent to the server;
- utilize wide fetches and wide inserts to batch `FETCH` or `INSERT` requests together into a single network packet, or set of packets.

For a network with relatively poor throughput[11], one can improve network efficiency in the following ways [26]:

- consider increasing the server's (or connection's) packet size to the largest value that makes sense for the network. This is specified through the the server's `-p` command line option or the client's `CommBufferSize` connection parameter.
- Consider using the `ReceiveBufferSize` and `SendBufferSize` TCP/IP protocol options on both the client and the server.
- turn prefetching off to eliminate the transfer of results to the client when those additional result rows are not needed by the application.

Pitfall 11 (Inefficient network usage with the TDS wire protocol) Both the Open Client and JConnect APIs utilize the Sybase TDS wire protocol over TCP/IP. For statements that return result sets—SQL queries or stored procedures—TDS is designed to deliver the *entire* result to the application, rather than wait for the client to request result rows through `FETCH` requests. Hence TDS does not inherently support cursor operations, such as `UPDATE WHERE CURRENT OF`. Open Client and JConnect, by default, do not support backwards-scrollable cursors at the TDS layer. Instead, scrollable cursors are implemented

10 You can approximate the network latency between two machines by the round trip time reported by SQL Anywhere's PING utility. The round trip time is the latency to transfer a data packet from one machine to a second machine plus the latency to transfer an acknowledgement from the second machine back to the first.

11 One way to measure network throughput is by copying a relatively large file from one machine to a second machine and timing the copy.

by caching the result set on the client. However, even if scrollable cursors are desired, SQL Anywhere does not support TDS scrolling requests, and consequently the client must still cache result rows to mimic scrollability.

As another example, the TDS specification does not include wide-`INSERT` capability. Instead, JConnect will simulate wide inserts by batching multiple `INSERT` statements together with an array of host variables: however, these `INSERT` statements are still executed at the server one statement at a time.

In other words, while the Open Client or JConnect application program interfaces may support cursor operations, the network traffic flow that actually occurs over the wire with such requests will be substantially different, and almost always considerably less efficient. Packet 'sniffing' software, such as Ethereal, can be used to trace and diagnose the network activity that results from such application requests. □

Pitfall 12 (Using DBISQL as a client program)
It can be tempting to use the DBISQL administration tool to serve as a client for performance testing, because one can not only use it to visualize test results but also to diagnose performance issues by generating graphical plans. However, DBISQL is not designed for benchmarking, for several reasons. Firstly, DBISQL can submit a `SELECT` statement to the server multiple times, once to `DESCRIBE` the result set, and the second to execute it. Secondly, while graphical plans with actual run-time statistics can be useful to diagnose optimization strategies, the monitors placed into the execution plan by the server to compute the run-times of plan operators consume additional CPU, approximately doubling the per-row CPU cost of retrieving a row from a page resident in the buffer pool and therefore skewing the query's performance.

The FETCHTST and TRANTEST utility programs are good alternatives to DBISQL for simulating client application behaviour. □

Summary. For SQL Anywhere applications, performance factors related to client configuration that could be considered in a performance evaluation include:

- which underlying communication protocols are utilized;
- prefetch settings;
- settings for the `ReceiveBufferSize` and `SendBufferSize` TCP/IP protocol options;
- network packet size.

Systematic experimental study of these factors, along with database server performance factors such as buffer pool sizes, are the subject of the next section.

5 Experiment design and analysis

Once the workload has been defined and validated, the next step in a performance evaluation is to define the experiments that will help in discovering the performance limitations the evaluation is intended to find. Unfortunately, all too often this step is oversimplified so that the 'benchmark' consists of a single experiment, possibly repeated multiple times as bottlenecks (some of them outlined in the previous sections) are discovered along the way. The problem with a single experiment is that it offers only a single data point—without repeating experiments systematically, an assessment of experimental error cannot be made. More importantly, a single test scenario yields only a single test result. Without additional testing scenarios, it is impossible to develop a *performance model* to assist in answering questions such as:

1. What are the most significant *performance factors* (e.g. disk subsystem characteristics, CPU speed, number of CPUs, buffer pool size, server multiprogramming level) that affect my application's performance? Which are the least significant?

2. Is it possible to achieve adequate performance for a given workload with fewer hardware requirements?

3. Which performance factors interact and must be considered in tandem when determining their impact on my application in a production setting?

To be able to answer these questions, a set of experiments must be designed to extract the necessary information from the performance evaluation. This must be done carefully so as to maximize useful information and simultaneously minimize the number of experiments. To perform comparative evaluations, it is necessary that each experiment be *repeatable* [13] so that it is possible to determine the effects of varying one or more performance factors, together or in isolation, and to determine the impact of experimental error.

5.1 Experimental designs

In the context of performance experiment design, an experiment's *response variable* is the metric used to measure system performance. With database application performance evaluation, this metric is typically the response time of a certain class of application request, or the overall system throughput in transactions-per-second. A *factor,* as described above, is a variable that affects the value of the response variable in a test. The values that a factor can assume are termed *levels.* For example, a database cache factor might have three levels for a suite of performance tests, such as 200MB, 600MB, and 1200MB, corresponding to small, medium, and large buffer pool configurations. Factors that are important to an experiment—that is, that impact the response variable measurably—are termed *primary factors*. Other factors that need not be measured because their impact on the response variable is minimal are termed *secondary factors*.

Kutner et al.[22] identify several broad classes of experimental designs:

1. single-factor designs,
2. complete and incomplete block designs,
3. nested designs,
4. repeated measures designs, and
5. factorial designs.

In a simple, single-factor experiment, the basic idea is to re-run a particular configuration multiple times, each time varying one performance factor at a time in order to determine that factor's effect on performance. The limitation of simple experiments is that they are designed only to test the effects of one factor at a time, and cannot analyze the *interaction effects* of related factors [20, pp. 278–9] [22, pp. 815–6]. With database performance analysis, *factorial designs* are often used because they provide insight into how performance factors interact with each other. In *full factorial* or *partial factorial* designs, combinations of factors are modified together, which result in a combinatorial explosion in the number of required experiments.

Example 4 (Factor combinations with factorial designs)
Suppose we wish to test all possible combinations of five performance factors where three factors have two levels, and the other two factors have three levels. This experimental design would require

$$2 \times 2 \times 2 \times 3 \times 3 = 72 \text{ experiments.}$$

If each experiment is repeated six times to determine the variance of experimental error, then a total of $72 \times 6 = 432$ experimental runs are required. □

The advantage of a full factorial design is that every combination of factor levels is analyzed, and their various interactions can be quantified and assessed. However, exhaustively testing this combinatorial explosion of factor combinations is rarely attempted due to its prohibitive cost. Rather, only selected combinations (termed a *fractional factorial design*) are usually evaluated: one can either reduce the number of factors considered, reduce the number of levels for each factor, or test only specific combinations rather than all possibilities. Often, all three simplifications are used to reduce the total number of experiments. Kutner et al. recommend three levels per factor, generally [22, pp. 650]. Jain

[20, pp. 280] strongly recommends reducing the number of levels of each factor to 2 because the set of 2^k experiments is easier to analyze statistically. Eliminating factors from consideration, or testing only certain combinations, must be considered carefully, as one must examine precisely *which* combinations of factors are the most relevant to the goals of the evaluation [22, pp. 648–652].

Whether two, three, or four levels per factor, one must choose the particular *values* of the levels with care. Kutner et al. [22, pp. 650] argue that the choice of levels for each factor is one of the most important experimental design decisions. For a 2^k experimental design it is tempting to choose low and high 'extreme' values to test a 'broad' performance range for a particular factor, but in a complex system there is no guarantee that system behaviour at the mid-point between the upper and lower values is similar in any way to the performance at either end of the range [22, pp. 650–1].

5.2 SQL Anywhere performance factors

In any benchmark experiment of a database application, the maximum value of the response variable will be constrained by the existence of any bottlenecks, whether they be bottlenecks within the application's design or contention within the database server itself. As outlined in previous sections, precisely how these bottlenecks influence overall system performance depends on many factors: application design (locking behaviour, scalability, client/server protocol); database design (schema design, availability of indexes, database page size); physical hardware (I/O architecture, number of processors, CPU speed, size and speed of on-chip cache), and database server configuration (multiprogramming level, database cache size, prefetch buffer size, and so on). All of these factors are important. Some of these factors tend to be highly variable and system-dependent, such as cache size, while others tend to be relatively static across installations of the same schema, such as database page size. With SQL Anywhere, the seven typical primary performance factors are, in no particular order:

1. the server multiprogramming level, as controlled by the `-gn` server option (default is 20 in most environments);
2. the size of the database buffer pool[12], as controlled by the `-c, -cl,` and `-ch` server options;
3. the database's page size;

12 SQL Anywhere supports dynamic buffer pool sizing, where the cache size is varied to match server requirements and the needs of other applications. While this self-management feature is often essential in embedded application environments, the variability of the buffer pool can lead to difficulties in performance analysis, particularly the repeatability of experiments. Consequently, we recommend performance analysis using fixed-sized buffer pools.

4. the number of CPUs available to the database server, as controlled by the `-gt` and `-gtc` server options;
5. the size of the experimental database instance, which influences the size of the buffer pool working set, assuming that the instance is representative of an actual database;
6. the speed and configuration of the server machine disk subsystem, in particular the number of disk arms available;
7. the size of the overall workload, which includes the transaction interarrival rate, the number of connections, and the mix of statements within the workload.

In any given situation, the relative importance of each of these factors to the goals of the performance evaluation may vary significantly. For example, if one is conducting a performance assessment to determine how the application may scale on the same hardware, at least two of the factors listed above (disk configuration and number of CPUs) need not be taken into account, considerably simplifying the analysis. Varying the workload, however, should be undertaken with care to avoid the pitfalls discussed previously.

5.3 Quantifiable performance assessment

Single measures of performance, such as mean response time, offer some information about the performance characteristics of a given experiment, but fail to provide a broad picture of overall performance. In the case of response time, for example, a better approach is not only to report the mean but also the variance of the distribution of transaction response time, which can be reported alternatively using the 90th or 95th percentile [32, pp. 557]. The accuracy, or *confidence interval,* of the reported value depends on the number of times the experiment is run and can be computed through statistical analysis techniques [20, pp. 216–220]. This analysis is much simpler if the number of samples N is sufficiently large—typically 30 or greater—and the size of the population the sample is based on is at least $2 \times N$. As an example, if one collected response time statistics from 30 clients in a synthetic, interactive workload that utilized 100 clients in total, then the results of the 30 measured clients could be interpreted using the *central limit theorem* [28], and the sampling statistics of various measures (mean, variance, proportions) taken during the experiment can be approximated by a normal or Gaussian distribution [13, 20, 21]. If the central limit theorem does not apply, then one must determine if the population can be approximated by a 'heavy-tailed' distribution [11], such as one that models Zipf's Law [30].

5.3.1 Establishing lower and upper bounds of performance

One useful tool in a performance evaluation is to establish a *baseline measurement.* In an empirical test of a workload developed using a trace of a production application, one could establish a baseline by executing the SQL trace of a single connection against the server with a user think time of zero. The results of this experiment would then represent a *lower bound* on the experimental results, in addition to providing a useful validation tool for the results of other multi-user experiments. With various assumptions, including that the access plans for each request are both optimal and consistent, and also that the server has reached a *steady state*, the results of this experiment would illustrate the best possible response time for the entire trace. This same approach could also be used in a piecemeal fashion to establish best-case performance measurements for individual queries within the workload; these best-case times could be subsequently used to analyze the times in a multi-user test, to assist in determining the cause of performance bottlenecks.

5.4 $2^k r$ experimental designs

The point of constructing a performance model through experimentation is twofold. Firstly, one usually desires to know whether or not a specific system configuration can 'handle' a particular workload. Secondly, and perhaps as importantly, one would like to determine the importance of any particular factor relative to the others, as measured by the proportion of the total variation in the response variable explained by each factor. For example, if the size of the buffer pool, and the number of physical disks, explain 90% and 5% respectively of the variation in overall system throughput (the response variable), then the number of disks may be considered unimportant in various situations that arise in practice *for such workloads.*

EXAMPLE 5 (HYPOTHETICAL 2^2 EXPERIMENT)
In a capacity planning study, we would like to determine the relative effect of database cache size and the server multiprogramming level on the performance of a given application workload, on the same physical hardware, as measured by total throughput in transactions per second (the response variable). To simplify the analysis, we establish two levels for each of the two primary factors as follows:

Factor	Level 0	Level 1
Multiprogramming Level A	20	30
Database cache size B	600 MB	1200 MB

The experimental outcomes appear in Table 3. □

Cache size	Multiprogramming level	
	20	30
600 MB	8.2	7.25
1200 MB	8.4	8.65

Table 3: Results of a 2^2 experiment, results stated in transactions-per-second.

Cache size	Multiprogramming level	
	20	30
600 MB	9.4	10.7
1200 MB	12.8	16.7

Table 4: Re-tested 2^2 experimental results with a modified workload.

At first glance, the results of Table 3 appear to indicate that the SQL Anywhere server does not scale well with this particular workload; doubling the database cache resulted only in a marginal improvement in overall system throughput, from 8.2 TPS to 8.4 TPS. On the other hand, increasing the multiprogramming level from 20 to 30 resulted in negative scalability: overall throughput decreased.

These results warrant investigation on at least two fronts. Firstly, the marginal improvement in throughput with a larger cache size requires explanation. One possibility is that Level 0 for the database buffer pool size is, in fact, sufficient for the given workload, and doubling the buffer pool size, therefore, has little effect on overall performance. Secondly, the negative scalability that resulted from increasing the multiprogramming level by 50% needs to be understood. There are several potential causes: one is that the increased multiprogramming level alters the memory quota for each request (see Section 4.2) so that the query optimizer is choosing more memory-efficient plans, but which have correspondingly greater execution times. Another possibility is that the increase in the number of concurrently-executing requests is increasing lock contention, therefore reducing the overall system throughput by increasing each request's elapsed time.

Table 4 reflects a complete re-test of the experimental results with a modified workload that addresses the problematic results discovered through detailed investigation of the individual experiments illustrated in Table 3. Note that these experimental results are merely *hypothetical,* and do not reflect the performance of any actual system. With the modifications to the workload, overall performance has improved, and the negative scalability at the higher multiprogramming level has been eliminated. The next step is an analysis of these results to construct a model that will identify the relative importance of the two factors tested. Moreover, the model may be useful in determining a focus for investigation of specific test results.

5.4.1 Statistical analysis of 2^2 experimental designs

A common technique used to model a set of experimental or observational results is *linear regression.* A *multiple linear regression model* can predict a response variable y as a linear function of k factors using an equation of the following form:

$$y = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_k x_k + e \tag{5}$$

where each b_i are the $k+1$ (fixed) coefficients and e is an error term. For the moment, we will avoid discussion of the computation of e and concentrate only on the determination of the coefficients b_i. In matrix notation, the linear regression model is

$$y = Xb + e \tag{6}$$

where

- b is a column vector with $k+1$ elements $\{b_0, b_1, b_2, \ldots, b_k\}$.
- y is a column vector of the n response variables $y = \{y_0, y_1, y_2, \ldots, y_n\}$.
- X is an n row by $k+1$ column matrix, whose $(i, j+1)$ element $X_{i,j+1} = 1$ if $j = 0$, and otherwise is the value of x_{ij}.

Solving the set of linear equations to find the values of b is then

$$b = (X^T X)^{-1}(X^T y) \tag{7}$$

At present this matrix equivalence is unimportant. However, we will see later that this matrix equivalence is useful when analyzing experiments of more than two factors.

A 2^2 experimental design is a special case of a 2^k design with $k = 2$, and the simplicity of the 2^2 design makes the computation of the linear coefficients relatively straightforward (a more precise mathematical treatment can be found in references [20] and [22]).

In the approach outlined below, we will utilize *indicator variables* both to simplify the analysis and to permit the use of both quantitative and qualitative factors [20, pp. 284–286]. We define two variables X_A and X_B as follows:

$$X_A = \begin{cases} -1 & \text{if the multiprogramming level is 20} \\ +1 & \text{if the multiprogramming level is 30} \end{cases}$$
$$X_B = \begin{cases} -1 & \text{if the cache size is 600 MB} \\ +1 & \text{if the cache size is 1200 MB} \end{cases} \tag{8}$$

An equivalent regression model to equation (5) is of the form

$$y = q_0 + q_A X_A + q_B X_B + q_{AB} X_A X_B. \tag{9}$$

Substituting the four observations from the experiments in Table 4, and replacing variables X_A and X_B with their corresponding values $\{1, -1\}$ for each experiment, we get the following four linear equations:

$$\begin{aligned} y_1 &= 9.4 = q_0 - q_A - q_B + qAB \\ y_2 &= 10.7 = q_0 + q_A - q_B - qAB \\ y_3 &= 12.8 = q_0 - q_A + q_B - qAB \\ y_4 &= 16.7 = q_0 + q_A + q_B + qAB. \end{aligned} \tag{10}$$

Solving this system of equations for each q_j, we get

$$\begin{aligned} q_0 &= \tfrac{1}{4}(y_1 + y_2 + y_3 + y_4) \\ q_A &= \tfrac{1}{4}(-y_1 + y_2 - y_3 + y_4) \\ q_B &= \tfrac{1}{4}(-y_1 - y_2 + y_3 + y_4) \\ q_{AB} &= \tfrac{1}{4}(y_1 - y_2 - y_3 + y_4). \end{aligned} \tag{11}$$

From our example above, the experimental results in Table 4 yield the coefficients $q_0 = 12.4$, $q_A = 1.3$, $q_B = 2.35$, and $q_{AB} = 0.65$, giving the regression equation

$$y = 12.4 + 1.3X_A + 2.35X_B + 0.65X_A X_B. \tag{12}$$

This result can be interpreted in the following way. The mean TPS of the set of experiments is 12.4 TPS. The effect of increasing the multiprogramming level by 50% is 1.3 TPS, while the effect of doubling the cache size is 2.25 TPS. The interaction between multiprogramming level and cache size resulted in an increase of 0.65 TPS. One can substitute the appropriate values (1 or -1) for each of X_A and X_B to determine a projected y-value based on the linear regression. As an example, suppose we have $X_A = -1$ and $X_B = 1$, corresponding to an experiment where the multiprogramming level is 20 and the database cache size is 1200 MB. These substitutions yield a y-value of $12.4 - 1.3 + 2.35 - 0.65 = 12.8$, identical to the measured value for the experiment as illustrated in Table 4. If instead we have $X_A = 1$ and $X_B = -1$, corresponding to an experiment where the multiprogramming level is 30 and the database cache size is 600 MB, we get a y-value of $12.4 + 1.3 - 2.35 - 0.65 = 10.7$, identical to the measured value.

Sign table method. We can utilize the equivalent matrix notation (Equation 6), rather than solving a system of linear equations, to construct a 'sign table' of the various coefficients, making the calculations more straightforward [20]. This will become important when we need to consider the interaction of k factors in Section 5.4.3 below.

Experiment	I	A	B	AB	y
1	1	-1	-1	1	9.4
2	1	1	-1	-1	10.7
3	1	-1	1	-1	12.8
4	1	1	1	1	16.7
	49.60	5.2	9.4	2.6	Total
	12.4	1.3	2.35	0.65	Total/4

TABLE 5: Sign-table method for a 2^2 experimental design.

One can determine the values of q for a 2^2 design by constructing a 4×4 sign table, as shown in Table 5. The first column of the table, labelled I (for *Identity*), consists of all '1's. Columns A and B contain values so that every possible permutation of 1 and -1 appear in each row, such that the sum of each of these columns is 0. The 1 and -1 values in the table correspond to the X-value assignments described earlier. Column AB contains the product of columns A and B in that row. Column y contains the response variable value corresponding to the factor levels chosen for A and B.

Once the sign table is constructed—a spreadsheet program, such as Microsoft Excel, is an ideal tool to use—the values in the 'Total' row are then computed by multiplying each value in column I by the corresponding value in column y and summing the four results. Column multiplication for columns A, B, and AB is carried out in the same fashion[13]. The effects of each factor are then easy to determine, simply by taking 1/4 of the total in each column. For column I, this value is simply the mean of the response variable y.

Allocation of variation. We would now like to quantify the importance of each factor used in the evaluation. Importance is measured by the proportion of the variation in the response variable (in Example 5, the system throughput in transactions-per-second) that factor is responsible for. To compute the sample variance s^2 of the n experimental results (the y values), we would use the standard formula:

$$s_y^2 = \frac{\sum_{i=1}^{2^2} (y_i - \bar{y})^2}{(n-1)} \tag{13}$$

where $\bar{y}$ denotes the mean. Note that the numerator, the 'sum of squares total' or SST, includes the total variation of y; however, in a 2^2 experimental de-

13 If using Microsoft Excel, this column multiplication is performed by the `SumProduct` function.

sign, the variation stems from each factor independently *and, additionally, to the interaction of the two.* Hence we have

$$\mathrm{SST} = \mathrm{SSA} + \mathrm{SSB} + \mathrm{SSAB} \tag{14}$$

and, with a little algebra, we have

$$\mathrm{SST} = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_{AB}^2 \tag{15}$$
$$= 4(q_A^2 + q_B^2 + q_{AB}^2) \tag{16}$$

or, in concrete terms with our example,

$$\mathrm{SST} = 6.76 + 22.09 + 1.69 = 30.54.$$

The importance of each factor can then be expressed as a ratio between the variation due to that factor over the total variation (sum of squares, or SST). Using the numbers from our example, the multiprogramming level (factor A) explains only 6.76/30.54 or 22.1 percent of the variation in the experimental results, while the database cache size (factor B) explains 22.09/30.54 or 72.33 percent of the variation.

5.4.2 Non-linear regressions

The analysis of a linear regression model, and any conclusions drawn from it, are meaningful only if the relationship between the response variable and the factors is linear. An excellent way to judge the nature of the relationship between the measured factors and the response variable is using a scatter plot[14]. Examples of scatter plot diagrams for a hypothetical 2^3 experiment design appear in Figure 2.

If the relationship between the values is not linear, it may be possible to model the relationship with a non-linear function, and subsequently transform it into a linear one. Such models are termed *colinear regressions.* For example, we may have the exponential regression

$$y = bx^a. \tag{17}$$

Taking the natural logarithm of both sides, we get

$$\ln y = \ln b + a \ln x \tag{18}$$

and $\ln y$ and $\ln x$ are now related with a linear function. Similar algebraic transformations can be made with a variety of other forms of functions, though the nature of that function may be difficult to determine. These transformations include square root transformations, arc sine transformations, and power

14 Note that with k factors, for $k > 2$, the scatter plot will be multi-dimensional.

transformations. Deciding which transformations to utilize may require complex analysis—further details about *curvilinear regression* and function transformations can be found in reference [20, Section 15.4].

Somewhat more problematic than non-linear association between the factors is when the effect of each factor is *multiplicative* rather than *additive*—it is additive relationships that are assumed by linear regression analysis. In other words, if the effects of two factors a and b multiply each other, the model is

$$y = ab \tag{19}$$

and this relationship is inappropriate to be modelled directly by linear regression. However, by taking the natural logarithm of both sides, we get

$$\ln y = \ln a + \ln b \tag{20}$$

and, if modelled in this fashion, one can apply the antilog to the additive effects to determine the multiplicative effects.

PITFALL 13 (MULTIPLICATIVE EFFECTS)
In a benchmark, or in a capacity planning exercise, multiplicative relationships are an easy trap to fall into. One example from Jain [20, pp. 304] is an experimental design whose factors include workload size and CPU speed. In a simple system, the interrelationship of these two factors is clearly multiplicative. In a more complex system, such as a database application benchmark, the effects are not necessarily multiplicative nor additive, but are related with a more complex function. This is because CPU speed often has an indirect effect on database application performance: as mentioned above, *all* CPUs *wait at the same speed*. □

In some situations, a straightforward analysis of the factors being considered will lead to the abandonment of an additive model. In addition, there are several indicators that can announce the potential for multiplicative effects. One indicator is a large ratio between the maximum and minimum values of the response variable y across the test spectrum. When this ratio is large—for example, an order of magnitude greater than the maximum value or higher—a logarithmic transformation should be considered. Another indicator of a log transformation is when the residuals are of the same order of magnitude as the response variable, or when a quantile-quantile plot of the residuals does not follow a normal distribution [20, pp. 304–8].

5.4.3 Statistical analysis of 2^k experimental designs

In the previous section, we described, in an abbreviated fashion, a linear regression model that represents the results of a 2^2 experimental design. We now generalize that model to determine the effect of k factors, each with two levels. Analyzing k factors with 2 levels each requires 2^k experiments.

Example 6 (Hypothetical 2^k experiment)
In this example, we modify the $k = 2$ experimental design to one where $k = 3$, adding the number of client connections as the additional factor. We assume that other secondary factors, such as the characteristics of the server hardware, remain constant. Our capacity planning study, now, is to determine the relative effect of database cache size, the server multiprogramming level, and the number of client connections on overall system performance, which again will be measured by total throughput in transactions/second (the response variable).

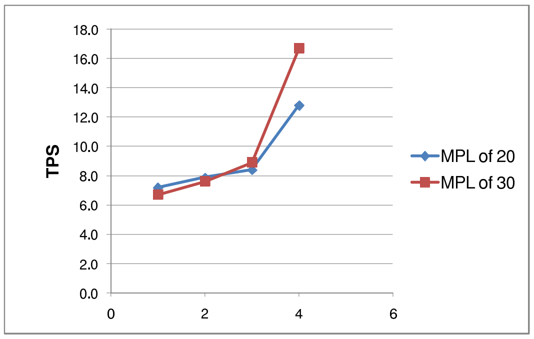
Factor	Level 0	Level 1
Multiprogramming Level A	20	30
Database cache size B	600 MB	1200 MB
Number of client connections C	100	200

The hypothetical experimental outcomes for the $2^3 = 8$ experiments appear in Table 6. □

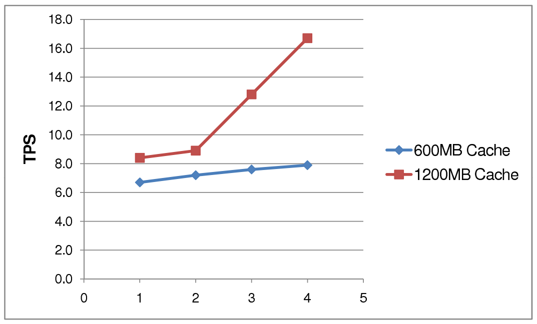
Pitfall 14 (Multicollinearity)
When considering 2^k and other classes of experimental designs, it may be tempting to include as many factors in the analysis as possible. One pitfall with high values of k is that the number of experiments required increases exponentially. However, another common issue with linear regression analysis is *multicollinearity*, or correlation between two or more factors [20, pp. 253-4]. In Example 6, the addition of client connections to the set of factors is problematic; should the size of the database instance, for example, also be included as a primary factor for analysis? In Section 4.3.2 above, we discussed the importance of carefully crafting the workload to properly model real-world conditions. It is likely that by doubling the number of client connections, a representative database would increase in size correspondingly. This correlation may lead to contradictory significance conclusions if the two factors were both considered in the same analysis, and their (measured) correlation was significant. □

Linear regression analysis of 2^k designs is a generalization of the analysis described above for 2^2 experimental designs. As with 2^2 designs, Equation (5) represents the multiple linear regression model. The additional complexity is that with k factors we have multiple combinations of interactions to consider: for $k > 2$, we have k main effects, $\binom{k}{2}$ two-factor interactions, $\binom{k}{3}$ three-factor interactions, and so on. Fortunately, with 2^k experimental designs, we can utilize the sign table method outlined previously to solve more easily the set of linear equations to determine the model's coefficients.

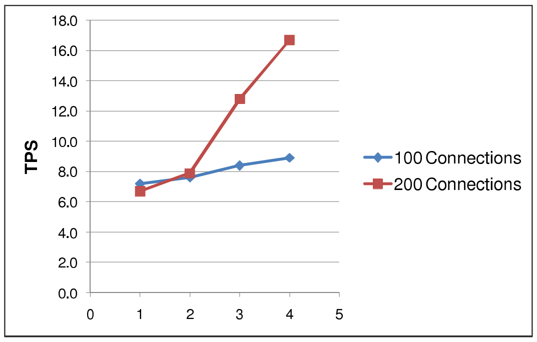
In addition to the variables X_A and X_B that represent the server multiprogramming level and database cache size respectively (Equation 8), we require



(a) Scatter plot of TPS in relation to multiprogramming level.



(b) Scatter plot of TPS in relation to database cache size.



(c) Scatter plot of TPS in relation to number of connections.

FIGURE 2: Scatter plots of 2^3 experimental data.

Cache	MPL of 20		MPL of 30	
Size	100	200	100	200
(MB)	Connections	Connections	Connections	Connections
600	7.2	7.9	7.6	6.7
1200	8.4	12.8	8.9	16.7

Table 6: 2^3 experiment results, stated in transactions-per-second.

Exp.	I	A	B	C	AB	AC	BC	ABC	y
1	1	-1	-1	-1	1	1	1	-1	7.2
2	1	1	-1	-1	-1	-1	1	1	7.6
3	1	-1	1	-1	-1	1	-1	1	8.4
4	1	1	1	-1	1	-1	-1	-1	8.9
5	1	-1	-1	1	1	-1	-1	1	7.9
6	1	1	-1	1	-1	1	-1	-1	6.7
7	1	-1	1	1	-1	-1	1	-1	12.8
8	1	1	1	1	1	1	1	1	16.7
	76.2	3.6	17.4	12	5.2	1.8	12.4	5	Total
	9.525	0.45	2.175	1.5	0.65	0.225	1.55	0.625	Total/2^3

Table 7: Sign-matrix method for a 2^3 experimental design.

another variable to represent the additional factor C, the number of client connections:

$$X_C = \begin{cases} -1 & \text{if 100 client connections} \\ +1 & \text{if 200 client connections.} \end{cases} \tag{21}$$

We then construct a sign table to represent the $2^3 = 8$ results in the same fashion as we constructed the sign table for the 2^2 case. The resulting sign table for this hypothetical example is shown in Table 7.

In Table 7, the mean performance is 9.525 TPS, and the effect of the three factors (multiprogramming level, cache size, and number of connections) taken in isolation are $q_A = 0.45$, $q_B = 2.175$, and $q_C = 1.5$, respectively. Interestingly, in this set of experiments the effect of the interaction of cache size and number of connections (column BC) is 1.55, which is greater than the effect of the number of connections alone. The sum-of-squares total, or SST, for this experimental design is

$$\begin{aligned}\text{SST} &= 2^3(q_A^2 + q_B^2 + q_C^2 + q_{AB}^2 + q_{AC}^2 + q_{BC}^2 + q_{ABC}^2) \\ &= 8(0.2025 + 4.73 + 2.25 + 0.4225 + 0.05 + 2.4 + 0.39) = 83.595. \qquad (22)\end{aligned}$$

The portion of variation allocated to each factor is then:

Factor	SST Component	Portion of Variation
Multiprogramming Level A	1.62	$1.62/83.595 = 1.93\%$
Database cache size B	37.85	$37.85/83.595 = 45.27\%$
Number of connections C	18	$18/83.595 = 21.53\%$
AB	3.38	4.04%
AC	0.405	0.004%
BC	19.22	22.99%
ABC	3.125	3.73%

Hence we conclude from this particular study that increasing the server multiprogramming level from 20 to 30 has a relatively insignificant effect on this particular workload. This confirms what is graphically shown in Figure 2(a): the two lines representing the response variable (TPS) for 100 and 200 client connections are nearly identical. However, for this workload the size of the database buffer pool accounts for 45% of the variation in performance, while the number of connections accounts for just under 22%. Together, those two factors account for an additional 23% of the variation in the performance results. Hence one can conclude that buffer pool size is a significant factor in server performance for this workload.

5.4.4 Analysis of $2^k r$ experimental designs

The idea behind $2^k r$ experimental designs is to repeat each experiment r times so that an estimate of experimental error can be made. The experimental error in a linear regression model is denoted by the e term in Equation (5).

PITFALL 15 (ASSESSING EXPERIMENTAL ERROR)
The merits of repeating experiments to assess experimental error are often downplayed. The typical argument against repeating experiments, other than their cost, is that the software under test is deterministic, and should yield identical results with each experiment. The flaw in this argument is that in a complex system, it is difficult to anticipate all potential software interactions, particularly those that depend on dates or time-of-day, which are common in business applications. □

A straightforward way to utilize r repeated experiments with the sign-matrix method is to take the mean $\bar{y}_i$ of each of the r observations of y_{ij} for each of the

2^k experiments, where y_{ij} denotes the jth replication of the ith experiment. Using $\bar{y_i}$ as the response variable for each of the jth experiments, a linear regression model for a 2^2 experiment design yields an *expected* response value $\hat{y_i}$ as follows:

$$\hat{y_i} = q_0 + q_A X_{A_i} + q_B X_{B_i} + q_{AB} X_{A_i} X_{B_i} \tag{23}$$

when the factors A and B are at values X_{A_i} and X_{B_i}. The difference between each of the r observations and this expected value represents the experimental error:

$$e_{ij} = y_{ij} - \hat{y_i} = y_{ij} - q_0 - q_A X_{A_i} - q_B X_{B_i} - q_{AB} X_{A_i} X_{B_i}. \tag{24}$$

Similar to the computation of the coefficients in the 2^2 case (see Equation 11), the formula to compute each q_j is [20, pp. 309]

$$q_j = \frac{1}{2^k} \sum_{i=1}^{2^k} S_{ij} \bar{y}_i \tag{25}$$

where S_{ij} denotes the entry in the sign table in row i, column j for that particular set of observations y_i for that experiment.

Allocation of variation. We can use Equation (24) to determine the experimental error for each of the $2^k r$ observations. By definition, the sum of the errors will be 0. We can compute the sum of the squared errors (SSE) to estimate the variance of the errors and confidence intervals for the effects:

$$\text{SSE} = \sum_{i=1}^{2^k} \sum_{j=1}^{r} e_{ij}^2 \tag{26}$$

and SSE is a component in our assessment of the allocation of variation:

$$\text{SST} = \text{SSA} + \text{SSB} + \text{SSAB} + \text{SSE}. \tag{27}$$

With a bit of algebra, similar to what was done for the 2^2 experimental design, the various formulas for determining SST and the other sum-of-squares are as follows [20, pp. 309]:

$$\text{SSY} = \sum_{i=1}^{2^k} \sum_{j=1}^{r} y_{ij}^2 \tag{28}$$

$$\text{SS0} = (2^k r) q_0^2 \tag{29}$$

$$\text{SST} = \text{SSY} - \text{SS0} \tag{30}$$

$$\text{SS}j = (2^k r) q_j^2 \text{ for all } j = 1, 2, \ldots, 2^k - 1 \tag{31}$$

$$\text{SSE} = \text{SST} - \sum_{j=1}^{2^k - 1} \text{SS}j \tag{32}$$

with the percentage of variation due to the jth effect computed as the ratio

$$\frac{\mathrm{SS}j}{\mathrm{SST}} \times 100\%. \tag{33}$$

By definition, the sum of the errors must add to 0, since the error is computed from the difference between each of the r observations of every experiment with the mean $\bar{y}_i$ of the response variables. Assuming that the errors are normally distributed, the variance of the errors for a $2^k r$ experimental design can be estimated from the SSE using the formula

$$s_e^2 = \frac{\mathrm{SSE}}{2^k(r-1)}. \tag{34}$$

Additional statistical tests based on these results can be computed, including the determination of confidence intervals for each of the effects. Jain [20, pp. 298] shows that because of the linear regression model, the variance of all of the effects is equal to $s_e^2/(2^k r)$, and hence the standard deviation of each effect is equal to $s_e/\sqrt{(2^k r)}$. Using Student's t-distribution, the confidence intervals for each of the effects are

$$q_j \pm t_{[1-\alpha/2;2^k(r-1)]} s_{q_j}. \tag{35}$$

5.5 Performance evaluation tools

A variety of performance evaluation tools are commercially available to assist in constructing a synthetic database application workload, and to execute various experiments and track their results. One of the more popular, though expensive, commercial options is the LoadRunner suite, offered by Mercury Software (*http://www.mercury.com*). At the opposite end of the spectrum, the `TRANTEST` utility program that comes with a standard SQL Anywhere installation offers an easy-to-use, albeit simplistic, way to implement a multi-user client-server performance evaluation. The drawbacks of `TRANTEST` include, among other characteristics, an unsophisticated mechanism for introducing user think time.

Sybase iAnywhere's consulting services group offers Floodgate, a tool that offers similar functionality to that of LoadRunner. Floodgate offers the ability to spawn multiple clients, executing different scripts, to mimic a complete interactive workload. Floodgate also supplies measurement facilities to measure and compute response times and other performance indicators.

In addition to the above tools, and the Sybase Central graphical administration tool included with SQL Anywhere, other methods are available to acquire additional performance statistics from a SQL Anywhere server. The most common of these, in a Microsoft Windows environment, is the use of the Windows NT Performance Monitor. The SQL Anywhere server process provides the Performance Monitor with various server- and database-specific performance counters, such as pages read per second, average number of active and queued requests, and so on. This information can be invaluable in diagnosing performance bottlenecks during evaluation, or in helping to validate the model workload. However, we caution that diagnosing performance bottlenecks is an extremely time-consuming task that requires considerable expertise. The project manager of any performance evaluation study must ensure that sufficient resources to perform this diagnosis is available when necessary.

Pitfall 16 (Interpreting Task Manager process statistics)
A common problem on Windows is identifying how much physical memory is in use by the database server. The Windows Task Manager, which lists all of the active processes on the system, does not actually report true memory usage in the 'Memory Size' column, but instead reports each task's *working set size*. Roughly speaking, on Windows XP and other similar Windows operating systems, working set size is the amount of physical RAM resident in memory and recently in use by the process.

The issue here is how Windows defines 'recently in use'. Windows periodically 'trims' the working set of a process by marking most or all of the memory allocated to it as 'not present'—as if it has been swapped out. However, Windows doesn't actually swap that memory out, although those memory pages will be among the first to be re-used for other purposes. If the process references that memory again, the process incurs a virtual memory fault. Windows catches this 'soft fault' and, knowing that the process is actually using that piece of memory, increments the process's working set accordingly. Typically, Task Manager will report a drastically reduced working set size for a process when that process is minimized: Windows is making an assumption—invalid for a database server—that a minimized application will utilize very little memory, and severely trims the working set size for that process. As a result, a process's working set size as displayed by Task Manager can fluctuate significantly, and neither it nor the 'VM Size' value reported by Task Manager are useful metrics to use in determining memory consumption of the server[15]. Instead, one should use the values reported for the 'Process/Private Bytes' and 'Process/Virtual Bytes' counters in Windows Performance Monitor to track server memory consumption and the size of the server's address space[16], respectively. On all platforms, you can look in the server window to see the cache size at startup, or `Select property('CacheSize')` to get the current cache

15 With Windows AWE extensions, the database cache pages are physically locked in memory and cannot be swapped out. Hence they are not considered part of the process's working set, and not reported by Task Manager. In fact, no builtin Windows utility exists to properly report the memory usage of a process that utilizes Address Windowing Extensions.

16 Neither of these values include memory for Address Windowing Extensions, either.

size.

A caveat: while using the Windows NT Performance Monitor can provide extremely useful information, its use can perturb the results of any performance evaluation because the NT Performance Monitor also consumes some degree of server resources. In addition, Performance Monitor statistics can be difficult to interpret properly without additional background knowledge. □

A comprehensive survey of commercially-available and freeware performance evaluation tools is beyond the scope of this paper.

6 Conclusions

Performance evaluation of any computer system is difficult, requiring considerable expertise. There are no hard-and-fast techniques of validating a workload model to ensure that it mimics the real world [14]. When a performance evaluation experiment indicates the possible existence of a performance bottleneck, it may take considerable time and effort to determine where the bottleneck is, and what, if anything, can be done to correct the problem.

In this whitepaper, we considered the important issues related to the construction of a benchmark workload, and described an analysis approach based on a 2^k experimental design. Other experimental designs are possible, including a 2^{k-p} design that embodies a fractional factorial experimental design. 2^{k-p} designs reduce the total number of experiments required, at the risk of *confounding* results, where the interaction between groups of factors cannot be adequately explained because the necessary detailed experimental results to determine that interaction are missing.

In other situations, it may be worthwhile to conduct an analysis using a *nested experimental design* or a *block design*. For example, a block design may be warranted when conducting experiments where one of the factors is the hardware platform and/or operating system. A block design may provide greater accuracy in the model because gross effects, such as memory allocation (an operating system trait) or underlying architecture (32-bit versus 64-bit), can cause significant variance in the experimental outcomes.

A description and analysis of 2^{k-p} and other more complex experimental designs is beyond the scope of this paper; the reader is encouraged to refer to references [20, 22] for additional background material.

Legal Notice

Contact Us

iAnywhere Solutions Worldwide Headquarters
One Sybase Drive, Dublin, CA, 94568 USA

Phone: 1-800-801-2069 (in US and Canada)
Fax: 1-519-747-4971
World Wide Web: http://www.ianywhere.com
E-mail: contact.us@ianywhere.com

References

[1] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, Barcelona, Spain, June 1998. IEEE Computer Society Press.

[2] Ramesh Bhashyam. TPC-D–the challenges, issues, and results. ACM SIGMOD *Record*, 25(4):89–93, December 1996.

[3] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th International Confer-*

ence on Very Large Data Bases, pages 8–19, Florence, Italy, October 1983. VLDB Endowment.

[4] Haran Boral and David J. DeWitt. A methodology for database system performance evaluation. In ACM SIGMOD *International Conference on Management of Data*, pages 176–185, Boston, Massachusetts, June 1984. Association for Computing Machinery.

[5] Peter M. Chen and David A. Patterson. A new approach to I/O performance evaluation–self-scaling I/O benchmarks, predicted I/O performance. ACM *Transactions on Computer Systems*, 12(4):308–339, November 1994.

[6] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. ACM *Transactions on Database Systems*, 9(2):163–186, 1984.

[7] Xilin Cui, Patrick Martin, and Wendy Powley. A study of capacity planning for database management systems with OLAP workloads. In *Proceedings of the International* CMG *Conference*, pages 515–526, Dallas, Texas, December 2003. Computer Measurement Group.

[8] Steven A. Demurjian, David K. Hsiao, Douglas S. Kerr, Robert C. Tekampe, and Robert J. Watson. Performance measurement methodologies for database systems. In *Proceedings of the 13th* ACM *Annual Conference*, pages 16–28, Denver, Colorado, October 1985. Association for Computing Machinery.

[9] Steven A. Demurjian, David K. Hsiao, and Roger G. Marshall. *Design Analysis and Performance Evaluation Methodologies for Database Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[10] David J. Dewitt, Shahram Ghandeharizadeh, and Donovan Schneider. A performance analysis of the gamma database machine. In ACM SIGMOD *International Conference on Management of Data*, pages 350–360, Chicago, Illinois, June 1988.

[11] Dror G. Feitelson. Workload modeling for performance evaluation. In Maria Carla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, Lecture Notes in Computer Science 2459, pages 114–141. Springer-Verlag, Rome, Italy, September 2002.

[12] Domenico Ferrari. *Computer Systems Performance Evaluation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[13] Domenico Ferrari, Giusepe Serazzi, and Alessandro Zeigner. *Measurement and Tuning of Computer Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[14] Paul J. Fortier and Howard E. Michel. *Computer Systems Performance Evaluation and Prediction.* Digital Press, Burlington, Massachusetts, 2003.

[15] G. Benton Gibbs, Jerry M. Enriquez, Nigel Griffiths, Corneliu Holban, Eunyoung Ko, and Yohichi Kurasawa. IBM E-*server pSeries Sizing and Capacity Planning: A Practical Guide.* IBM Corporation, San Jose, California, March 2004. IBM Redbook Order Number SG–247071, available at http://www.redbooks.ibm.com/redbooks/pdfs/sg247071.pdf.

[16] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan-Kaufmann, San Francisco, California, second edition, 1993.

[17] Seungrahn Hahn, M.H. Ann Jackson, Bruce Kabath, Ashraf Kamel, Caryn Meyers, Ana Rivera Matias, Merrilee Osterhoudt, and Gary Robinson. *Capacity Planning for Business Intelligence Applications: Approaches and Methodologies*. IBM Corporation, San Jose, California, November 2000. IBM Redbook Order Number SG24–5689, available at http://www.redbooks.ibm.com/redbooks/pdfs/sg245689.pdf.

[18] Richard A. Hankins, Trung A. Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th International Symposium on Microarchitecture*. IEEE Computer Society Press, December 2003.

[19] Paula B. Hawthorn and Michael Stonebraker. Performance analysis of a relational data base management system. In ACM SIGMOD *International Conference on Management of Data*, pages 1–12, Boston, Massachusetts, May 1979. Association for Computing Machinery.

[20] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, New York, 1991.

[21] K. [Krishna] Kant. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, New York, New York, 1992.

[22] Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. *Applied Linear Statistical Models*. McGraw-Hill International, New York, New York, fifth edition, 2005.

[23] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, Barcelona, Spain, June 1998. IEEE Computer Society Press.

[24] Winfried Materna. An approach to the construction of workload models. In M. Arató, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*, pages 161–177. North-Holland, Vienna, Austria, February 1979.

[25] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, San Jose, California, October 1994. Association for Computing Machinery.

[26] Ian McHardy. Optimizing Adaptive Server Anywhere performance over a WAN. Technical whitepaper, Sybase iAnywhere, Waterloo, Ontario, 2005. Available at `http://www.ianywhere.com/downloads/whitepapers/wan_tuning.pdf`.

[27] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[28] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Company, New York, New York, 1989.

[29] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul [Per-Åke] Larson. TPC-DS, Taking decision support benchmarking to the next level. In ACM SIGMOD *International Conference on Management of Data*, pages 582–587, Madison, Wisconsin, June 2002. Association for Computing Machinery.

[30] Viswanath Poosala. Zipf's law. Technical report, University of Wisconsin, Madison, Wisconsin, 1995. Available at `http://www.bell-labs.com/user/poosala/pub.html`.

[31] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, San Jose, California, October 1998. Association for Computing Machinery.

[32] Tom Sawyer. Doing your own benchmark. In Gray [16], pages 543–561.

[33] Dennis Shasha and Philippe Bonnet. *Database Tuning*. Morgan-Kaufmann, San Francisco, California, 2003.

[34] Standard Performance Evaluation Corporation, Warrenton, Virginia. SPEC *jAppServer2004 Benchmark Specification, Revision 1.08*, December 2006. Available at http://www.spec.org/jAppServer2004.

[35] G. C. Steindel and H. G. Madison. A benchmark comparison of DB2 and the DBC/1012. In *Proceedings, International Conference on Management and Performance Evaluation of Computer Systems*, pages 360–369, Orlando, Florida, 1987. The Computer Measurement Group.

[36] Liba Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. Elsevier, New York, New York, 1976.

[37] Transaction Processing Performance Council, San Jose, California. TPC *Benchmark* D *(Decision Support) Standard Specification, Revision 2.0.0.12*, June 1998.

[38] Transaction Processing Performance Council, San Jose, California. TPC *Benchmark* H *(Decision Support) Standard Specification, Revision 1.1.0*, June 1999.

[39] Transaction Processing Performance Council, San Jose, California. TPC *Benchmark* R *(Decision Support) Standard Specification, Revision 1.0.1*, February 1999.

[40] Ted J. Wasserman, Patrick Martin, and Haider Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proceedings of the 7th* ACM *International Workshop on Data Warehousing and* OLAP, pages 7–13, Wasington, D.C., November 2004. Association for Computing Machinery.

[41] Ted J. Wasserman, Patrick Martin, and Haider Rizvi. Sizing DB2 UDB servers for business intelligence workloads. In *Proceedings of the 2004 Conference of the* IBM *Centre for Advanced Studies on Collaborative Research*, pages 135–149, Markham, Ontario, October 2004. IBM Corporation.