1

Facilis Descensus Averni[1]

When we decided to write this book, we chose the topic "Tuning Client/Server SQL Programs" because we had a specific plan. Each word in the topic narrows the focus. Let's look at the main words.

Tuning means enhancing speed. It's possible to make a distinction between tuning and optimizing—tuning is what you do to a database (e.g., change cache size, repartition, rebuild indexes) while optimizing is what you do to a program (e.g., adjust queries, take advantage of existing resources, rewrite application code). But we do not make such a fine distinction in this book. As for *speed*, the word can mean two things: **response time** (the time a statement takes to execute) and **throughput** (the number of operations the DBMS can do in a time unit). Our concern here is mostly with throughput.

SQL is a language that is supported by a **Database Management System (DBMS)** implementation. We decided to use a generic term so you'll know that we're not talking about a particular brand—this book is about what's in common for all major brands. There really is a standard core that they all share, so we can be specific in our recommendations without having to be specialists. Where there are many differences between DBMSs, you'll see charts outlining

1. Literally, "the descent into Hell is easy" . . . it's getting back out again that's hard! From Virgil.

the differences. Where there are few differences, you'll see a discussion of what's normal and (if necessary) a sidebar about the exceptional case.

Programs are things you write in a language such as SQL, C, or Java. So don't expect to see long discussions about how to choose equipment, what flags to set when you install, or which tools you'll need to monitor the users. Those tasks properly belong to the **Database Administrator (DBA).** We know that much of this book will interest DBAs; we know that many readers are both DBAs and programmers. That's great. We only want to make it clear that, in this book, we are addressing the people who write applications.

This Subject Is Important

"In our experience (confirmed by many industry experts) 80% of performance gains on SQL Server come from making improvements in SQL code, not from devising crafty configuration adjustments or tweaking the operating system."

—Kevin Kline et al., *Transact-SQL Programming,* O'Reilly & Associates

"Experience shows that 80 to 90 per cent of all tuning is done at the application level, not at the database level."

—Thomas Kyte, *Expert One on One: Oracle,* Wrox Press

No matter which DBMS you use, you can enhance its performance by doing the right thing. Enhancing performance is a broad field. It includes:

- Coding SQL statements without doing things that everyone knows are counter-productive
- Understanding the physical structure of a typical database
- Solving real problems rather than imaginary ones

Let's take a trivial example. Suppose you have this SQL statement:

```
SELECT column1 FROM Table1
  WHERE column1 = 77
```

For this kind of statement, we believe the essential question is—Should there be an index on `column1`? So we've devoted a whole chapter to the subject of indexes—what they look like, what variations exist, how indexes affect data changes, and so on. In another chapter, we address the question of how to use EXPLAIN (or its equivalent) to find out whether your particular DBMS actually uses an index for a particular SELECT. That illustrates our priority—we think that the first priority is the concept: indexes. Certainly, though, we must also care about the method: diagnostic tools. We hope that, with the concepts firmly planted in your mind, you will quickly arrive at the right point. We don't recommend that you implement any idea in this book without testing it first—but without ideas, you'll flounder randomly between plans without knowing if your final choice really is the best one.

We think an idea is sound if performance improves by 5% or more on most DBMSs. That may appear to be a modest goal, but consider. First, we always test to ensure that the idea doesn't *harm* performance on some other DBMS—we believe an idea is only good when it applies universally. Second, we think that even a small number like 5% matters when an operation occurs many times for many rows. Third, we are asking you to read only once, because once you have the information, it's a tiny effort to reuse it for years. Fourth, the improvement often will be many times more than 5%. Fifth, effects may be small, but they're also cumulative.

We also hope that you find the topic, well, interesting. If it's any incentive at all, let us assure you that many database practitioners, and all the good ones, are fascinated by these two questions—How does it work? How ought it to work?

The Big Eight

"In theory there is no difference between theory and practice. But, in practice, there is."
—Jan L.A. van de Snepscheut

You don't have to specialize to find useful ideas in this book. Over and over again, we have found that basic matters that are true for DBMS #1 are also true for DBMS #2, DBMS #3, and so on, across the board. For example, we can say that "DBMSs store data in fixed-size pages and the size is a power of two." But isn't that a bold generalization? Let's be honest. We have not checked this statement on every DBMS that exists, we know it's not a law of nature, and in fact we know of at least two DBMSs for which the statement is false. But, in this book, when we make a claim for "all DBMSs" or just "DBMSs" we're not being vague or general—we mean, very specifically, eight particular DBMSs that we have actually tested and for which we guarantee that the statement is true at time of writing. We call these DBMSs "the Big Eight"—not a standard term, but a convenient way to direct your memory to this introductory explanation.

We chose the Big Eight according to the following criteria:

- The DBMS must be an SQL client/server DBMS. DBMSs of other types were excluded.
- The DBMS must have at least a 1% share of the market in North America and Europe according to published and verifiable surveys, or it must be widely known because it's open source.
- The DBMS must support **Java Database Connectivity (JDBC)** and **Open Database Connectivity (ODBC)**.

We want to emphasize that no DBMS got on our list due to its quality. We chose each of the Big Eight based only on the probability that you'll encounter it or something very much like it. Because the Big Eight have a combined market share of over 85%, with the open source DBMSs having been downloaded hundreds of thousands of times from Internet sites, we're confident that you'll be dealing with one of the Big Eight at least 90% of the time. Table 1–1 shows which DBMSs were tested for this book.

When researching the material for this book, we installed the latest versions of each DBMS available for the MS WindowsNT platform at the time of writing. Each DBMS was installed and tested using the default systems recommended in the vendors' instructions, except as indicated in the following section, "Installation Parameters." In some cases, we used evaluation copies or personal editions; in no case did we test with extra-cost or little-used options—our policy was to ignore nondefault installation options, settings, and switches.

Table 1–1 *The Big Eight*

Short Name	Product Name & Version	Remarks
IBM	IBM DB2 Universal Database 7.2	
Informix	IBM Informix Dynamic Server 9.3	Now owned by IBM
Ingres	Ingres II 2.5	Owned by Computer Associates International
InterBase	InterBase 6.0	Open source version. Owned by Borland Software Corporation
Microsoft	Microsoft SQL Server 2000	
MySQL	MySQL 3.23	Open source. Owned by MySQL AB
Oracle	Oracle 9i	
Sybase	Sybase ASE 12.5	

We have avoided endorsing or denouncing any vendor, because our object is to help you improve your SQL, given the hand you've been dealt.

There are test results throughout this book; however, there are no absolute performance figures or inter-DBMS comparisons. There are two reasons for this. One is obvious: such figures wouldn't fit the purpose of the book. The other is interesting: three of the Big Eight—Informix, Microsoft, and Oracle—have end-user license agreements that specifically prohibit publication of benchmarks.

Installation Parameters

As indicated earlier, in order to minimize our use of extra-cost or little-used options and to level the playing field between DBMSs as much as possible, we installed the Big Eight using the default systems recommended in the vendors' instructions except in the following cases:

- For the sake of consistency, we wanted to run all our tests using the same character set—the Windows 1252 code page—for every DBMS if at all possible. We chose this code page because we were testing on a Windows NT system and wanted to utilize international character set possibilities.
- We wanted to test the differences between dictionary and binary sorts for every DBMS if at all possible.

For IBM, these criteria meant that the IBM database was created with the default "IBM-1252" character set and default "Local Alphabet" dictionary sort order. We used CHAR columns to test dictionary sorts and CHAR FOR BIT DATA columns to test binary sorts. IBM doesn't provide SQL Standard-compliant CHARACTER SET or COLLATE options.

For Informix, these criteria meant that Informix was installed with the default "EN_US 8859-1" for client and server locales, and nondefault "EN_GB.8859-1" db_locale, which provides a dictionary sort order for NCHAR columns. We used NCHAR columns to test dictionary sorts and CHAR columns to test binary sorts. Informix doesn't provide SQL Standard-compliant CHARACTER SET or COLLATE options.

For Ingres, these criteria meant that the Ingres database was created with the default "WIN1252" character set and the nondefault "lmulti" dictionary sort order. We used CHAR columns to test dictionary sorts and BYTE columns to test binary sorts. Ingres doesn't provide SQL Standard-compliant CHARACTER SET or COLLATE options.

For InterBase, these criteria meant that the InterBase database was created with DEFAULT CHARACTER SET WIN1252. We used NCHAR columns with COLLATE EN_US to test dictionary sorts and NCHAR columns with no COLLATE clause to test binary sorts. We also used NCHAR columns with (a) COLLATE DA_DA to test Danish/Norwegian sorts, (b) COLLATE DE_DE to test German sorts, (c) COLLATE IS_IS to test Icelandic sorts, (d) COLLATE EN_UK to test Irish sorts, (e) COLLATE ES_ES to test Spanish sorts, and (f) COLLATE FI_FI and COLLATE SV_SV to test Swedish/Finnish sorts.

For Microsoft, these criteria meant that SQL Server was installed with the default "1252/ISO" Character Set and the default "Dictionary Order, case insensitive" Sort Order for CHAR columns, and a nondefault "Binary" Unicode Collation. We used CHAR columns with no COLLATE clause to test dictionary sorts and CHAR columns with COLLATE SQL_Latin1_General_BIN to test binary sorts. We also used CHAR columns with (a) COLLATE SQL_Danish to test Danish/Norwegian sorts, (b) COLLATE German_PhoneBook to test German phone book sorts, (c) COLLATE SQL_Icelandic to test Icelandic sorts, (d) COLLATE Mexican_Trad_Spanish and COLLATE Modern_Spanish to test Spanish sorts, and (e) COLLATE SQL_SwedishStd to test Swedish/Finnish sorts. (Note: Where applicable, PREF was always indicated, the code page was 1252, case sensitivity was CI, and accent sensitivity was AS.)

For MySQL, these criteria meant that MySQL was installed with the default "Latin1" (aka iso_1) character set. We used CHAR columns to test dictionary sorts and CHAR BINARY columns to test binary sorts. MySQL doesn't provide SQL Standard-compliant CHARACTER SET or COLLATE options.

For Oracle, these criteria meant that Oracle was installed with the default "WIN1252" character set. We used CHAR columns with NLS_SORT=XWEST_EUROPEAN to test dictionary sorts and CHAR columns with NLS_SORT=BINARY to test binary sorts. We also used CHAR columns with (a) NLS_SORT=DANISH and NLS_SORT=NORWEGIAN to test Danish/Norwegian sorts, (b) NLS_SORT=XGERMAN to test German dictionary sorts, (c) NLS_SORT=GERMAN_DIN to test German phone book sorts, (d) NLS_SORT=ICELANDIC to test Icelandic sorts, (e) NLS_SORT=XWEST_EUROPEAN to test Irish sorts, (f) NLS_SORT=XSPANISH to test Spanish Traditional sorts, (g) NLS_SORT=SPANISH to test Spanish Modern sorts, and (h) NLS_SORT=FINNISH to test Swedish/Finnish sorts.

For Sybase, these criteria meant that Sybase was installed with the nondefault "Character Set = iso_1" and nondefault "Sort Order = Dictionary." We used CHAR columns to test dictionary sorts and BINARY columns to test binary sorts. Sybase doesn't provide SQL Standard-compliant CHARACTER SET or COLLATE options.

Test Results

Throughout this book, you'll see sets of SQL statements that show a test we ran against the Big Eight. The second statement in each example includes a note at the bottom that states: *GAIN: x/8*. That's an important number. The gain shows how many of the Big Eight run faster when an SQL statement is optimized by implementing the syntax shown in the second example. We recorded a gain for a DBMS if performance improved by 5% or greater. Mileage varies with different data and different machines, of course. We're only reporting what our tests showed.

"GAIN: 0/8" means "you'd be wasting your time if you rearranged this particular SQL statement into optimum order because the DBMS does this for you." "GAIN: 4/8," on the other hand, means that half of the Big Eight performed better when the suggested syntax was used, while the other half executed both statements equally well. Even if the gain is only 1/8 (meaning only one of the Big Eight improved on the second statement), you'll be better off using our suggested syntax because this means you'll improve performance some of the time without ever harming performance the rest of the time. That is, none of our suggestions will cause a performance decline on any of the Big Eight—with one exception.

The exception is that, in a few cases, one DBMS showed aberrant behavior and declined in performance on the second statement, while all the rest showed impressive gains. In such cases, we felt the possible improvement was worthy of mention anyway. Each exception notes the DBMS with which you should *not* use our improved syntax.

All tests were run on a single CPU Windows NT machine, with no other jobs running at the same time. The main test program was written in C and used ODBC calls to communicate with the DBMSs. A second test program used JDBC to test the DBMSs' response to specific calls. Unless otherwise indicated, each performance test was run three times for 10,000 rows of randomly inserted data, with the relevant column(s) indexed as well as without indexes. The gain for each DBMS was then calculated as the average of the three test runs.

We want to emphasize that our gain figures do *not* show absolute performance benchmark results. That is, a "GAIN: 4/8" note does not mean that any or all DBMSs ran 50% faster. It merely means that 50% of the DBMSs ran faster, and the rest showed no change.

Portability

As a bit of a bonus, because we had access to all these DBMSs, we will also be able to give you some information about portability.

We regard portability as a matter of great importance. In the first place, this book is about client/server applications. We anticipate that you will want to write code that works regardless of DBMS. In fact, we anticipate that you may not even know for which DBMS you are coding!

To avoid depending on any vendor's idiosyncrasies, all SQL examples and descriptions of SQL syntax in this book are written in standard SQL—that is, ANSI/ISO SQL:1999—whenever possible. When standard SQL omits a feature but it's common to all DBMSs—for example, the CREATE INDEX statement—our examples use syntax that will run on most platforms. Where nonstandard and uncommon syntax exists or had to be tested, we have identified it as such. Look for our "Portability" notes; they indicate where syntax other than standard SQL might be required.

To aid you, we've also added comparison charts that highlight differences between the SQL Standard and the Big Eight. In these tables, you'll sometimes see "N/S" in the ANSI SQL row. This means that the SQL Standard considers the feature to be implementation-defined; that is, there is no Standard-specified requirement. Instead, the decision on how to implement the feature is made by the DBMS vendor.

Optimizing SQL for all dialects is different from tuning for a single package. But the coverage in this book is strictly the universal stuff.

Terminology and Expectations

We expect that you're a programmer coding for (or about to begin coding for) an SQL DBMS. Because of this, we won't be explaining basic SQL syntax or programming techniques. Our assumption is that you already know basic SQL syntax, how to program with an SQL **Application Programming Interface (API)** such as ODBC or JDBC, how to write stored procedures, are familiar with how indexes operate, and so on. We also expect that you're familiar with SQL terms as used in introductory SQL texts. For example, suppose we illustrate a SELECT statement like this:

```
SELECT <select list>
  WHERE <search conditions>
  FROM <Table list>
  GROUP BY <grouping columns>
    HAVING <conditions>
  ORDER BY <sorting columns>
```

We assume you've already seen terms like "select list" and "search condition" and "grouping column," and we won't repeat their well-known definitions. At most, we'll just provide a brief refresher for common SQL syntax and concepts.

Some further terms are not so well known, but are important for understanding this book. We will define such terms the first time we use them. If you miss such a definition, you can look it up in the glossary in Appendix B.

Conventions

We use a particular style in our examples. SQL keywords are always in uppercase (e.g., SELECT). Table and other major SQL object names are initial capitalized (e.g., `Table1`, `Index1`); column names are in lowercase (e.g., `column1`). When it is necessary to use more than one line, each line will begin with a clause-leader keyword.

We deliberately avoid "real-world" names like `Employees` or `cust_id` because we believe that meaningful names would distract from the universality of the example. Sometimes, though, when illustrating a particular characteristic, we will use a name that hints at the item's nature. For example:

```
SELECT column1, column2
  FROM Table1
  WHERE indexed_column = <literal>
```

This book doesn't contain many SQL syntax diagrams, but here's a very brief refresher on the common variant of **Backus-Naur Form (BNF)** notation that we've used:

- < >

  Angle brackets surround the names of syntactic elements. Replace the names with real data.

- [ ]

  Square brackets surround optional syntax. You may either use or omit such syntax.

- { }

  Braces surround mandatory syntax groups. You must include one of the options for the group in your SQL statement.
- |

  The vertical bar separates syntactic elements. You may use only one of the options in your SQL statement.

Generalities

"Beware of entrance to a quarrel; but being in,
Bear't that th'opposed may beware of thee."
—William Shakespeare, *Hamlet*

We can start off with some general tips.

SQL is a procedural language. Despite the confusion and outright lies about this point, it is a fact that an SQL statement's clauses are processed in a fixed order. And despite the set orientation of SQL, the DBMS must often operate on an SQL result set one row at a time. That is, if you're executing this statement:

```
UPDATE Table1
   SET column1 = 5
   WHERE column2 > 400
```

SQL's set orientation means the DBMS will determine that, for example, six rows meet the condition that requires them to be updated. After that, though, the DBMS must actually change all six of the rows—one row at a time.

The relational model is inherently efficient. Dr. Codd's rules and the subsequent work on **normalization** are based on the proven truism that mathematical foundations lead to solid structures. (Normalization is the process of designing a database so that its tables follow the rules specified by relational theory.)

Always assume you will do a **query** at least 100 times. That will make you ask yourself whether you want to use a procedure, a view, a trigger, or some other object and might also make you ask—If this is so common, has someone already written something like it?

And now, let's get started.