9

Tuning Database Design for High Performance

9.1 Introduction ................................................................. 9-1

9.2 Underlying Principles .................................................... 9-2
What Databases Do • Performance Spoilers

9.3 Best Practices ............................................................... 9-3
Tuning Hardware • Tuning the Virtual Machine • Tuning
the Operating System • Tuning Concurrency
Control • Indexes • Tuning Table Design

9.4 Tuning the Application Interface ...................................... 9-10
Assemble Object Collections in Bulk • Art of Insertion

9.5 Monitoring Tools ......................................................... 9-12

9.6 Tuning Rules of Thumb .................................................. 9-13

9.7 Summary and Research Results ....................................... 9-13

9.8 Information .................................................................. 9-13

Glossary ........................................................................... 9-14

Philippe Bonnet
IT University of Copenhagen

Dennis Shasha
New York University

9.1 Introduction

In fields ranging from arbitrage to sensor processing, speed of access to data can determine success or failure. Database tuning is the activity of making a database system run faster. Like optimization activities in other areas of computer science and engineering, database tuning must work within certain constraints. Just as compiler optimizers, for example, cannot directly change the underlying hardware but can change register allocations, database tuners cannot change the underlying database management system software, but can make use of the options it offers.

The database administrator in charge of tuning can, for example, modify the design of tables, select new indexes, rearrange transactions, tamper with the operating system, or buy hardware. The goals are to increase throughput and reduce response time. In our personal experience, tuning efforts can have dramatic effects, e.g., reduce a query time from nine hours to 15 seconds.

Further, interactions between database components and the nature of the bottlenecks change with technology. For example, hard drives have been used as secondary storage since the advent of relational database systems; nowadays, only solid state drives (SSD) offer the high-performance throughput needed to match the speed of processing units in a balanced system. In SSDs, sequential IOs are not so much faster than random IOs (which is the case for disks).

Tuning, then, is for well-informed generalists. This chapter introduces a principled foundation for tuning, focusing on principles that have been robust for years and promise to still hold true for years to come. We include experiments from specific commercial and free database systems, which will become obsolete much faster.
9.2 Underlying Principles

To understand the principles of tuning, you must understand the two main kinds of database applications and what affects performance.

9.2.1 What Databases Do

At a high level of abstraction, databases are used for two purposes: on-line transaction processing and decision support. On-line transaction processing typically involves access to a small number of records, generally to modify them. A typical such transaction records a sale or updates a bank account. These transactions use indexes to access their few records without scanning through an entire table. E-commerce applications are modern examples of OLTP applications. It seems that potential e-customers will abandon a site if they have to wait more than for a google search.

Decision support queries, by contrast, read many records often from a data warehouse, compute an aggregate result, and sometimes apply that aggregate back to an individual level. Typical decision support queries are “find the total sales of widgets in the last quarter in the northeast” or “calculate the available inventory per unit item.” Sometimes the results are actionable as in “find frequent flyer passengers who have encountered substantial delays in their last few flights and send them free tickets and an apology.”

Data mining is, in practice, best done outside the database management system, though it may draw samples from the database. In so doing, it issues decision support queries.

9.2.2 Performance Spoilers

Having divided the database applications into two broad areas, we can now discuss what slows them down.

1. Random vs. sequential disk accesses: On hard disk drives, sequential disk bandwidth is between one and two orders of magnitude (10–100 times) greater than random-access disk bandwidth. On SSDs, there is no intrinsic difference between the throughput of sequential and random reads because read performance depends only on the degree of parallelism achieved within the SSD. Depending on the SSD model, the relative performance of sequential and random reads will vary, but not much. Thus on hard drives, many sequential IOs might be much faster than few random IOs; while on SSDs fewer IOs is always better. Concretely, index accesses tend to be random whereas scans are sequential. Thus, on hard disks, removing an index may sometimes improve performance, because the index usually performs random reads and behaves poorly. While on SSDs, indexed accesses are better than scans provided they result in fewer IOs.

2. Lack of parallelism: Modern computer systems support parallelism in processing (e.g., multicore and/or graphics processors), storage (controllers, disks, and SSDs). Poor layout and operating system choices may inhibit parallelism.

3. Imprecise data searches: These occur typically when a selection retrieves a small number of rows from a large table, yet must search the entire table to find those data. Establishing an index may help in this case, though other actions, including reorganizing the table, may also have an effect.

4. Many short data interactions, either over a network or to the database: This may occur, for example, if an object-oriented application views records as objects and assembles a collection of objects by accessing a database repeatedly from within a “for” loop rather than as a bulk retrieval.

5. Delays due to lock conflicts.

These occur either when update transactions execute too long or when several transactions want to access the same datum, but are delayed because of locks. A typical example might be a single variable...
that must be updated whenever a record is inserted. In the following example, the **COUNTER** table contains the next value which is used as a key when inserting values in the **ACCOUNT** table.

```sql
begin transaction
   NextKey: = select nextkey from COUNTER;
   insert into ACCOUNT values (nextkey, 100, 200);
   update COUNTER set nextkey = NextKey + 1;
end transaction
```

When the number of such transactions issued concurrently increases, **COUNTER** becomes a bottleneck because all transactions read and write the value of next key. This problem is only amplified by the need to achieve a high degree of parallelism.

As mentioned in the introduction, avoiding performance problems requires changes at all levels of a database system. We will discuss tactics used at several of these levels and their interactions—hardware, concurrency control subsystem, indexes, and conceptual level. There are other levels such as recovery and query rewriting that we mostly defer to Ref. [4].

### 9.3 Best Practices

Understanding how to tune each level of a database system requires understanding the factors leading to good performance at that level. Each of the following subsections discusses these factors before discussing tuning tactics.

#### 9.3.1 Tuning Hardware

Each processing unit consists of one or more processors, one or more disks, and some memory. Assuming a high-end 180-GIPS (billion instructions per second) processor, the CPU will become the bottleneck for on-line transaction-processing applications when it is attached to 1 high-end SSDs, 10 mid-range SSDs, or 7,200 hard disks (counting 500,000 instructions per random IO issued by the database system and 400,000 random IO per seconds on a high-end SSD, 40,000 random IO per second on a mid range SSD, and 50 random IO per second on a high-end hard disk).

Decision-support queries, by contrast, often entail massive scans of a table. In theory, a 180-GIPS processor is saturated when connected to 1 high-end SSD, 5 mid-range SSDs, or 36 hard disks (counting 500,000 instruction per sequential IO and 1 million sequential IO per second for high-end SSD, 100,000 sequential IO per second for mid-range SSDs, and 10,000 IO per second for hard disks). A few years ago, the system bus was the bottleneck when reading sequentially from several disks. Nowadays, a commonplace PCIe system bus of 5 GT/s (Giga Transfers per second) is not saturated by the aggregated bandwidth of 10 SSDs. The problem is to find enough connectors on the system bus.

Note that a 180 GIPS processor relies on tens of hardware threads to deliver such throughput. Each core delivers typically 20–30 GIPS. Also the figures mentioned earlier assume that the IO submission rate is high enough to leverage the internal parallelism of the SSDS. Achieving a high IO submission rate is the key objective when tuning a database system on SSD.

Summing up, decision-support sites may need fewer disks per processor than transaction-processing sites for the purposes of matching aggregate disk bandwidth to processor speed.* Note that storing 1 TB of data on hard disk is an order of magnitude cheaper than storing 1 TB of data on a SSD.

---

* This point requires a bit more explanation. There are two reasons you might need more disks: (1) for disk bandwidth (the number of bytes coming from the disk per second); or (2) for space. Disk bandwidth is usually the issue in on-line transaction processing. Decision support applications tend to run into the space issue more frequently, because scanning allows disks to deliver their optimal bandwidth.
All applications will benefit from a mix of solid state drives for high-performance access and hard disks for inexpensive storage.

Random access memory (RAM) obviates the need to go to disk. Database systems reserve a portion of RAM as a buffer. In all applications, the buffer usually holds frequently accessed pages (hot pages, in database parlance) including the first few levels of indexes. Increasing the amount of RAM buffer tends to be particularly helpful in on-line transaction applications where disks are the bottleneck, particularly for smaller tables.

The read hit ratio in a database is the portion of database reads that are satisfied by the buffer. Hit ratios of 90% or higher are common in on-line transaction applications, but less common in decision support applications. Even in transaction processing applications, hit ratios tend to level off as you increase the buffer size if there is one or more tables that are accessed unpredictably and are much larger than available RAM (e.g., sales records for a large department store).

The decreasing cost of RAM makes it economical to attach 256 GB of RAM to a CPU. As a consequence, mid-sized databases can be manipulated entirely in RAM centrally on a single server, or in a distributed manner across a cluster. From a tuning point of view, the main goal is then to minimize the number of buffer reads on each server and the amount of communication across servers.

9.3.2 Tuning the Virtual Machine

It is today commonplace to rely on virtual machines to manage the hardware resources in a company. As a result, database systems often run within a virtual machine. Tuning the virtual machine to match the needs of the database system has become an important task. While configuring the number of CPUs, the size of the RAM or the number of disks associated to a virtual machine is rather straightforward. The configuration of the disk controllers and of the disk characteristics is a bit more subtle:

- No host caching: The database system expects that the IOs it submits are transferred to disk as directly as possible. It is thus critical to disable file system caching on the host for the IOs submitted by the virtual machine.
- Hardware supported IO virtualization: A range of modern CPUs incorporate components that speed up access to IO devices through direct memory access and interrupt remapping (i.e., Intel’s VT-d and AMD’s AMD-vi). Such a feature is important to allow databases executing within a virtual machine to leverage the performance characteristics of high-end SSDs.
- Statically allocated disk space: When creating a virtual disk, it is possible to reserve the space physically on disk (static allocation). Alternatively, the virtual disk grows dynamically as required. Static allocation is preferable for database systems. First, the overhead of space reservation is paid only once at disk creation time, not while executing transactions. Second, static allocation guarantees that a disk is mapped on contiguous physical space. This is especially important when using hard disks, because contiguous allocation guarantees that the sequential IOs submitted by a database system on a virtual disk are actually mapped onto sequential IOs on disk.

9.3.3 Tuning the Operating System

The operating system provides the storage and processing abstractions to the database system. More specifically, file system and thread management impact database performance.

9.3.3.1 Thread Management

Processor affinity refers to binding of a given software thread to a specific hardware thread. It allows efficient cache reuse, and avoids context switches. For transactional workloads, the log write can be assigned to a specific hardware thread to guarantee that it can continuously write to the log.
Modern database systems allow a database administrator to dynamically change the priority of the different threads running in the system. Also, the database system implements mechanisms that avoid the priority inversion problem, where a high-level transaction waits for a low-level transaction to release a resource it is waiting for.

9.3.4 Tuning Concurrency Control

As the chapter on Concurrency Control and Recovery in this handbook explains, database systems attempt to give users the illusion that each transaction executes in isolation from all others. The ANSI SQL standard, for example, makes this explicit with its concept of degrees of isolation [3,5]. Full isolation or **serializability** is the guarantee that each transaction that completes will appear to execute one at a time *except that its performance may be affected by other transactions*. This ensures, for example, that in an accounting database in which every update (sale, purchase, etc.) is recorded as a double-entry transaction, any transaction that sums assets, liabilities, and owners’ equity will find that assets equal the sum of the other two. There are less stringent notions of isolation that are appropriate when users do not require such a high degree of consistency.

The concurrency-control algorithm in predominant use is two-phase locking, sometimes with optimizations for data structures. Two-phase locking has **read** (or **shared**) and **write** (or **exclusive**) locks. Two transactions may both hold a shared lock on a datum. If one transaction holds an exclusive lock on a datum, however, then no other transaction may hold any lock on that datum; in this case, the two transactions are said to **conflict**. The notion of datum (the basic unit of locking) is deliberately left unspecified in the field of concurrency control, because the same algorithmic principles apply regardless of the size of the datum, whether a page, a record, or a table. The performance may differ, however. For example, record-level locking works much better than table-level locking for on-line transaction processing applications.

**Snapshot isolation** avoids read locks as it gives each transaction the illusion that it accesses, throughout its execution, the state of the database that was valid when it started. With snapshot isolation read transactions do not conflict with write transactions [9]. However, snapshot isolation does not ensure serializability. For example, consider a transaction T1 that reads value x and writes that value into y. Suppose that T2 reads y and writes that value into x. If x is initially 3 and y is initially 17, then any serial execution will guarantee that x and y are the same at the end. Under snapshot isolation T1 may read x at the same time that T2 reads y, in which case x will have the value 17 at the end and y will have the value 3.

9.3.4.1 Rearranging Transactions

Tuning concurrency control entails trying to reduce the number and duration of conflicts. This often entails understanding application semantics. Consider, for example, the following code for a purchase application of item i for price p for a company in bankruptcy (for which the cash cannot go below 0):

```
Purchase Transaction (p,i)
1 Begin Transaction
2 if cash < p then roll back transaction
3 inventory(i): = inventory(i) + p
4 cash: = cash – p
5 End Transaction
```

From a concurrency-control-theoretical point of view, this code does the right thing. For example, if the cash remaining is 100, and purchase P1 is for item i with price 50, and purchase P2 is for item j with price 75, then one of these will roll back.

From the point of view of performance, however, this transaction design is very poor, because every transaction must acquire an exclusive lock on cash from the beginning to avoid deadlock.
(Otherwise, many transactions will obtain shared locks on cash and none will be able to obtain an exclusive lock on cash.) That will make cash a bottleneck and have the effect of serializing the purchases. Since inventory is apt to be large, accessing inventory\(i\) will take at least one disk access, taking about 5 ms. Since the transactions will serialize on cash, only one transaction will access inventory at a time. This will limit the number of purchase transactions to about 50 per second. Even a company in bankruptcy may find this rate to be unacceptable.

A surprisingly simple rearrangement helps matters greatly:

**Redesigned Purchase Transaction \((p,i)\)**

1. Begin Transaction
2. \(\text{inventory}(i) := \text{inventory}(i) + p\)
3. if cash < \(p\) then roll back transaction
4. else cash := cash − \(p\)
5. End Transaction

Cash is still a hot spot, but now each transaction will avoid holding cash while accessing inventory. Since cash is so hot, it will be in the RAM buffer. The lock on cash can be released as soon as the commit occurs.

Advanced techniques are available that “chop” transactions into independent pieces to shorten lock times further. We refer interested readers to Ref. [4].

### 9.3.4.2 Living Dangerously

Many applications live with less than full isolation due to the high cost of holding locks during user interactions. Consider the following full-isolation transaction from an airline reservation application:

**Airline Reservation Transaction \((p,i)\)**

1. Begin Transaction
2. Retrieve list of seats available.
3. Reservation agent talks with customer regarding availability.
4. Secure seat.
5. End Transaction

The performance of a system built from such transactions would be intolerably slow, because each customer would hold a lock on all available seats for a flight while chatting with the reservations agent. This solution does, however, guarantee two conditions: (1) no two customers will be given the same seat, and (2) any seat that the reservation agent identifies as available in view of the retrieval of seats will still be available when the customer asks to secure it.

Because of the poor performance, however, the following is done instead:

**Loosely Consistent Airline Reservation Transaction \((p,i)\)**

1. Retrieve list of seats available.
2. Reservation agent talks with customer regarding availability.
3. \(\text{BEGIN TRANSACTION}\)
4. Secure seat.
5. \(\text{END TRANSACTION}\)

This design relegates lock conflicts to the secure step, thus guaranteeing that no two customers will be given the same seat. It does allow the possibility, however, that a customer will be told that a seat is available, will ask to secure it, and will then find out that it is gone. This has actually happened to a particularly garrulous colleague of ours.
9.3.5 Indexes

Access methods, also known as indexes, are discussed in another chapter. Here we review the basics, then discuss tuning considerations. An index is a data structure plus a method of arranging the data tuples in the table (or other kind of collection object) being indexed. Let us discuss the data structure first.

9.3.5.1 Data Structures

Two data structures are most often used in practice: B-trees and Hash structures. Of these, B-trees are used the most often (one vendor’s tuning book puts it this way: “When in doubt, use a B-tree”). Here, we review those concepts about B-trees most relevant to tuning.

A B-tree (strictly speaking a B + tree) is a balanced tree whose nodes contain a sequence of key–pointer pairs [2]. The keys are sorted by value. The pointers at the leaves point to the tuples in the indexed table).

B-trees are self-reorganizing through operations known as splits and merges (though occasional reorganizations for the purpose of reducing the number of seeks do take place). Further, they support many different query types well: equality queries (find the employee record of the person having a specific social security number), min–max queries (find the highest-paid employee in the company), and range queries (find all salaries between $70,000 and $80,000).

Because an access to disk secondary memory costs about 5 ms if it requires a seek (as index accesses will), the performance of a B-tree depends critically on the number of nodes in the average path from root to leaf. (The root will tend to be in RAM, but the other levels may or not be, and the farther down the tree the search goes, the less likely they are to be in RAM.) The number of nodes in the path is known as the number of levels. One technique that database management systems use to minimize the number of levels is to make each interior node have as many children as possible (1000 or more for many B-tree implementations). The maximum number of children a node can have is called its fanout. Because a B-tree node consists of key–pointer pairs, the bigger the key is, the lower the fanout.

For example, a B-tree with a million records and a fanout of 1000 requires three levels (including the level where the records are kept). A B-tree with a million records and a fanout of 10 requires 7 levels. If we increase the number of records to a billion, the numbers of levels increase to 4 and 10, respectively. This is why accessing data through indexes on large keys is slower than accessing data through small keys on most systems.

Hash structures, by contrast, are a method of storing key–value pairs based on a pseudorandomizing function called a hash function. The hash function can be thought of as the root of the structure. Given a key, the hash function returns a location that contains either a page address (usually on disk) or a directory location that holds a set of page addresses. That page either contains the key and associated record or is the first page of a linked list of pages, known as an overflow chain leading to the record(s) containing the key. (You can keep overflow chaining to a minimum by using only half the available space in a hash setting.)

In the absence of overflow chains, hash structures can answer equality queries (e.g., find the employee with Social Security number 156-87-9864) in one disk access, making them the best data structures for that purpose. The hash function will return arbitrarily different locations on key values that are close but unequal, e.g., Smith and Smythe. As a result, records containing such close keys will likely be on different pages. This explains why hash structures are completely unhelpful for range and min–max queries.

9.3.5.2 Clustering and Sparse Indexes

The data structure portion of an index has pointers at its leaves to either data pages or data records.

- If there is at most one pointer from the data structure to each data page, then the index is said to be sparse.
- If there is one pointer to each record in the table, then the index is said to be dense.
If records are small compared to pages, then there will be many records per data page and the data structure supporting a sparse index will usually have one less level than the data structure supporting a dense index. This means one less disk access if the table is large. By contrast, if records are almost as large as pages, then a sparse index will rarely have better disk access properties than a dense index.

The main virtue of dense indexes is that they can support certain read queries within the data structure itself in which case they are said to cover the query. For example, if there is a dense index on the keywords of a document retrieval system, a query can count the records containing some term, e.g., “derivatives scandals,” without accessing the records themselves. (Count information is useful for that application, because queriers frequently reformulate a query when they discover that it would retrieve too many documents.) A secondary virtue is that a query that makes use of several dense indexes can identify all relevant tuples before accessing the data records; instead, one can just form intersections and unions of pointers to data records.

A clustering index on an attribute (or set of attributes) \(X\) is an index that puts records close to one another if their \(X\)-values are near one another. What “near” means depends on the data structure. On B-trees, two \(X\)-values are near if they are close in their sort order. For example, 50 and 51 are near, as are Smith and Sneed. In hash structures, two \(X\)-values are near only if they are identical.

Some systems such as Oracle and InnoDB use an implicit form of clustering called an index organized table. This is a table that is clustered on its primary key (or a system-generated row id, if there is no primary key).

Sparse indexes must be clustering, but clustering indexes need not be sparse. In fact, clustering indexes are sparse in some systems (e.g., SQL Server, ORACLE hash structures) and dense in others (e.g., ORACLE B-trees, DB2). Because a clustering index implies a certain table organization and the table can be organized in only one way at a time, there can be at most one clustering index per table.

A nonclustering index (sometimes called a secondary index) is an index on an attribute (or set of attributes) \(Y\) that puts no constraint on the table organization. The table can be clustered according to some other attribute \(X\) or can be organized as a heap, as we discuss later. A nonclustering index must be dense, so there is one leaf pointer per record. There can be many nonclustering indexes per table.

A heap is the simplest table organization of all. Records are ordered according to their time of entry. That is, new insertions are added to the last page of the data structure. For this reason, inserting a record requires a single page access. Reading a record requires a scan.

A table and the indexes associated with it might be partitioned. Each partition is associated to a file and each file is associated to a disk. This way a table might be accessed in parallel. Indexes are defined on each partition. The dispatching of tuples into partitions is either based on round-robin, on ranges defined on a partitioning attribute, or on a hash function applied on a partitioning attribute.

Nonclustering indexes are useful if each query retrieves significantly fewer records than there are pages in the file. We use the word “significant” for the following reason: a table scan can often save time by reading many pages at a time, provided the table is stored on contiguous tracks. Therefore, on hard disk, if the scan and the index both read all the pages of the table, the scan may complete 100 times faster than if it read one page at a time. On SSD, this ratio is much smaller.

In summary, nonclustering indexes work best if they cover the query. Otherwise, they work well, especially on SSDs, if the average query using the index will access fewer records than there are data pages. Large records and high selectivity both contribute to the usefulness of nonclustering indexes. On hard disks, scans are hard to beat.

### 9.3.5.3 Data Structures for Decision Support

Decision support applications often entail querying on several, perhaps individually unselective, attributes. For example, “Find people in a certain income range who are female, live in California, buy climbing equipment, and work in the computer industry.” Each of these constraints is unselective in itself, but together form a small result. The best all-around data structure for such a situation is the bitmap.
A bitmap is a collection of vectors of bits. The length of each such “bit vector” equals the length of the table being indexed and has a 1 in position i if the ith record of the table has some property. For example a bitmap on state would consist of 50 bit vectors, one for each state. The vector for California would have a 1 for record i if record i pertains to a person from California. In our experiments, bitmaps outperform multidimensional indexes by a substantial margin.

Some decision support queries compute an aggregate, but never apply the result of the aggregate back to individuals. For example, you might want to find the approximate number of Californian women having the properties mentioned earlier. In that case, you can use approximate summary tables as a kind of indexing technique. The Aqua system [1], for example, proposes an approximation based on constructing a database from a random sample of the most detailed table T (sometimes known as the fact table in data warehouse parlance) and then joining that result with the reference tables R1, R2, …, Rn based on foreign key joins.

9.3.5.4 Final Remarks Concerning Indexes

The main point to remember is that the use of indexes is a two-edged sword: we have seen an index reduce the time to execute a query from hours to a few seconds in one application, yet increase batch load time by a factor of 80 in another application. Add them with care.

9.3.6 Tuning Table Design

Table design is the activity of deciding which attributes should appear in which tables in a relational system. The Conceptual Database Design chapter discusses this issue, emphasizing the desirability of arriving at a normalized schema. Performance considerations sometimes suggest choosing a nonnormalized schema, however. More commonly, performance considerations may suggest choosing one normalized schema over another or they may even suggest the use of redundant tables.

9.3.6.1 To Normalize or Not to Normalize

Consider the normalized schema consisting of two tables: Sale(sale_id, customer_id, product, quantity) and Customer(customer_id, customer_location).

If we frequently want sales per customer location or sales per product per customer location, then this table design requires a join on customer_id for each of these queries. A denormalized alternative is to add customer_location to Sale, yielding Sale(sale_id, customer_id, product, quantity, customer_location) and Customer(customer_id, customer_location). In this alternative, we still would need the Customer table to avoid anomalies such as the inability to store the location of a customer who has not yet bought anything.

Comparing these two schemas, we see that the denormalized schema requires more space and more work on insertion of a sale. (Typically, the data-entry operator would type in the customer_id, product, and quantity; the system would generate a sale_id and do a join on customer_id to get customer_location.) On the other hand, the denormalized schema is much better for finding the products sold at a particular customer location.

The tradeoff of space plus insertion cost vs. improved speeds for certain queries is the characteristic one in deciding when to use a denormalized schema. Good practice suggests starting with a normalized schema and then denormalizing sparingly.

9.3.6.2 Redundant Tables

The previous example illustrates a special situation that we can sometimes exploit by implementing wholly redundant tables. Such tables store the aggregates we want. For example:

Sale(sale_id, customer_id, product, quantity) Customer(customer_id, customer_location) Customer_Agg (customer_id, totalquantity) Loc_Agg (customer_location, totalquantity). This reduces the query time,
but imposes an update time as well as a small space overhead. The tradeoff is worthwhile in situations where many aggregate queries are issued (perhaps in a data warehouse situation) and an exact answer is required.

### 9.3.6.3 Tuning Normalized Schemas

Even restricting our attention to normalized schemas without redundant tables, we find tuning opportunities because many normalized schemas are possible. Consider a bank whose `Account` relation has the normalized schema (account_id is the key):

- `Account(account_id, balance, name, street, postal_code)`

Consider the possibility of replacing this by the following pair of normalized tables:

- `AccountBal(account_id, balance)`
- `AccountLoc(account_id, name, street, postal_code)`

The second schema results from **vertical partitioning** of the first (all nonkey attributes are partitioned). The second schema has the following benefits for simple account update transactions that access only the id and the balance:

- A sparse clustering index on account_id of `AccountBal` may be a level shorter than it would be for the `Account` relation, because the name, street, and postal_code fields are long relative to account_id and balance. The reason is that the leaves of the data structure in a sparse index point to data pages. If `AccountBal` has far fewer pages than the original table, then there will be far fewer leaves in the data structure.
- More account_id–balance pairs will fit in memory, thus increasing the hit ratio. Again, the gain is large if `AccountBal` tuples are much smaller than `Account` tuples.

On the other hand, consider the further decomposition:

- `AccountBal(account_id, balance)`
- `AccountStreet(account_id, name, street)`
- `AccountPost(account_id, postal_code)`

Though still normalized, this schema probably would not work well for this application, since queries (e.g., monthly statements, account update) require both street and postal_code or neither. Vertical partitioning, then, is a technique to be used for users who have intimate knowledge of the application.

In recent years, database systems have been designed as **column-stores**, where vertical partitioning is the rule and efficient mechanisms are provided for improving hit ratio both in the database cache and in CPU cache lines, and for minimizing IOs based on efficient column compression. Column stores are well suited for decision support applications in which tables may have hundreds of columns, but each query uses only a few. The benefit then is that only the few relevant columns need to be processed, saving both input/output time and processing time. Further, storing columns separately gives greater opportunities to achieve high compression.

### 9.4 Tuning the Application Interface

A central tuning principle asserts **start-up costs are high; running costs are low**. When applied to the application interface, this suggests that you want to transfer as much necessary data as possible between an application language and the database per connection. Here are a few illustrations of this point.
9.4.1 Assemble Object Collections in Bulk

Object-oriented encapsulation allows the implementation of one class to be modified without affecting the rest of the application, thus contributing greatly to code maintenance. Encapsulation sometimes is interpreted as "the specification is all that counts." Unfortunately, that interpretation can lead to horrible performance.

The problem begins with the fact that the most natural object-oriented design on top of a relational database is to make records (or sometimes fields) into objects. Fetching one of these objects then translates to a fetch of a record or a field. So far, so good.

But then the temptation is to build bulk fetches from fetches on little objects (the "encapsulation imperative"). The net result is to execute many small queries, each of which goes across the programming language to database system boundary, instead of one large query.

Consider, for example, a system that delivers and stores documents. Each document type (e.g., a report on a customer account) is produced according to a certain schedule that may differ from one document type to another. Authorization information relates document types to users. This gives a pair of tables of the form:

\[
\text{authorized} (\text{user, documenttype}) \\
\text{documentinstance} (\text{id, documenttype, documentdate})
\]

When a user logs in, the system should say which document instances he or she can see. This can easily be done with the join:

\[
\begin{align*}
\text{select } & \text{documentinstance.id, documentinstance.documentdate} \\
\text{from } & \text{documentinstance, authorized} \\
\text{where } & \text{documentinstance.documenttype} = \text{authorized.documenttype} \\
& \text{and authorized.user} = \text{<input user name>}
\end{align*}
\]

But if each document type is an object and each document instance is another object, then one may be tempted to write the following code:

```java
Authorized authdocs = new Authorized();
authdocs.init(<input user name>);
for (Enumeration e = authdocs.elements(); e.hasMoreElements();)
{
    DocInstance doc = new DocInstance();
    doc.init(e.nextElement());
    doc.print();
}
```

This application program will first issue one query to find all the document types for the user (within the init method of Authorized class):

\[
\begin{align*}
\text{select } & \text{documentinstance.documenttype} \\
\text{from } & \text{authorized} \\
& \text{where authorized.user} = \text{<input user name>}
\end{align*}
\]

and then for each such type t to issue the query (within the init method of DocInstance class):

\[
\begin{align*}
\text{select } & \text{documentinstance.id, documentinstance.documentdate} \\
\text{from } & \text{documentinstance} \\
& \text{where documentinstance.documenttype} = \text{t}
\end{align*}
\]

This is much slower than the previous SQL formulation. The join is performed in the application and not in the database server.
The point is not that object-orientation is bad. Encapsulation contributes to maintainability. The point is that programmers should keep their minds open to the possibility that accessing a bulk object (e.g., a collection of documents) should be done directly rather than by forming the member objects individually and then grouping them into a bulk object on the application side.

9.4.2 Art of Insertion

We have spoken so far about retrieving data. Inserting data rapidly requires understanding the sources of overhead of putting a record into the database:

1. As in the retrieval case, the first source of overhead is an excessive number of round trips across the database interface. This occurs if the batch size of your inserts is too small. In fact up to 100,000 rows, increases in the batch size improve performance on most systems.
2. The second reason has to do with the ancillary overhead that an insert causes: updating all the indexes on the table. Even a single index can hurt performance.
3. Finally, the layers of software within a database system can get in the way. Database systems provide bulk loading tools that achieve high performance by bypassing some of the database layers (mostly having to do with transactional recovery) that would be traversed if single row INSERT statements were used. For instance, SQL* Loader is a tool that bulk loads data into Oracle databases. It can be configured to bypass the query engine of the database server (using the direct path option).

The SQL Server BULK INSERT command and SQL*Loader allow the user to define the number of rows per batch or the number of kilobytes per batch. The minimum of the two is used to determine how many rows are loaded in each batch. There is a tradeoff between the performance gained by minimizing the transaction overhead in the omitted layers and the work that has to be redone in case a failure occurs.

9.5 Monitoring Tools

When your system is slow, you must figure out where the problem lies. Is it a single query? Is some specific resource misconfigured? Is there insufficient hardware? Most systems offer the following basic monitoring tools [6,7,8]:

1. Time-spent monitors capture for a transaction, or a session the time spent in the different database components. This information allows to identify for a transaction, which component can be tuned to significantly improve performance.
2. Event monitors (sometimes known as Trace Data Viewer or Server Profiler) capture usage measurements (processor usage ratio, disk usage, locks obtained, etc.) throughout the system at the end of each query. It is sometimes hard to differentiate the contribution of the different transactions to the indicators. Also, it is hard to evaluate how a given indicator impacts the performance of a specific transaction.
3. If you have found an expensive query, you might look to see how it is being executed by looking at the query plan. These Plan Explainer tools tell you which indexes are used, when sorts are done and which join ordering is chosen.
4. If you suspect that some specific resource is overloaded, you can check the consumption of these resources directly using operating system commands. This includes the time evolution of processor usage, disk queuing, and memory consumption. Dynamic tracing frameworks are allowed to collect detailed information even in the context of production systems.
9.6 Tuning Rules of Thumb

Often, tuning consists in applying the techniques cited earlier, such as the selection and placement of indexes or the splitting up of transactions to reduce locking conflicts. At other times, tuning consists in recognizing fundamental inefficiencies and attacking them.

1. Simple problems are often the worst. We have seen a situation where the database was very slow because the computer supporting the database was also the mail router. Offloading nondatabase applications is often necessary to speed up database applications.

2. Another simple problem having a simple solution concerns locating and rethinking specific queries. The authors have had the experience of reducing query times by a factor of 10 by the judicious use of outer joins to avoid superlinear query performance.

3. The use of triggers can often result in surprisingly poor performance. Since procedural languages for triggers resemble standard programming languages, bad habits sometimes emerge. Consider, for example, a trigger that loops over all records inserted by an update statement. If the loop has an expensive multitable join operation, it is important to pull that join out of the loop if possible. We have seen another 10-fold speedup for a critical update operation following such a change.

4. There are many ways to partition load to avoid performance bottlenecks in a large enterprise. One approach is to distribute the data across sites connected by wide-area networks. This can result, however, in performance and administrative overheads unless networks are extremely reliable. Another approach is to distribute queries over time. For example, banks typically send out 1/20 of their monthly statements every working day rather than send out all of them at the end of the month.

9.7 Summary and Research Results

Database tuning is based on a few principles and a body of knowledge. Some of that knowledge depends on the specifics of systems (e.g., which index types each system offers), but most of it is independent of version number, vendor, and even data model (e.g., hierarchical, relational, or object-oriented). This chapter has attempted to provide a taste of the principles that govern effective database tuning.

Various research and commercial efforts have attempted to automate the database tuning process [10]. Among the most successful is the tuning wizard offered by Microsoft’s SQL server. Given information about table sizes and access patterns, the tuning wizard can give advice about index selection among other features. Tuners would do well to exploit such tools as much as possible. Human expertise then comes into play when deep application knowledge is necessary (e.g., in rewriting queries and in overall hardware design) or when these tools do not work as advertised (the problems are all NP-hard).

Diagnosing performance problems and finding solutions may not require a good bedside manner, but good tuning can transform a slow and sick database into one full of pep.

9.8 Information

Whereas the remarks of this chapter apply to most database systems, each vendor will give you valuable specific information in the form of tuning guides or administrator’s manuals. The guides vary in quality, but they are particularly useful for telling you how to monitor such aspects of your system as the relationship between buffer space and hit ratio, the number of deadlocks, the input/output load, and so on.

Our book Database Tuning: Principles, Experiments, and Troubleshooting Techniques, published by Morgan Kaufmann goes into greater depth regarding all the topics in this chapter. Our website http://www.databasetuning.org contains a repository of experiments and results obtained on current database systems.
Glossary

**B-tree**: The most used data structure in database systems. A B-tree is a balanced tree structure that permits fast access for a wide variety of queries. In virtually all database systems, the actual structure is a B+ tree in which all key–pointer pairs are at the leaves.

**Clustering index**: A data structure plus an implied table organization. For example, if there is a clustering index based on a B-tree on last name, then all records with the last names that are alphabetically close will be packed onto as few pages as possible.

**Column-oriented store**: In such a storage layout, tables are laid out column-wise, so all the employee ids are contiguous, all the salaries are contiguous, etc.

**Conflict (between locks)**: An incompatibility relationship between two lock types. Read locks are compatible (nonconflicting) with read locks, meaning different transactions may have read locks on the same data item s. A write lock, however, conflicts with all kinds of locks.

**Covering index**: An index whose fields are sufficient to answer a query.

**Data mining**: The activity of finding actionable patterns in data.

**Decision support**: Queries that help planners decide what to do next, e.g., which products to push, which factories require overtime, and so on.

**Denormalization**: The activity of changing a schema to make certain relations denormalized for the purpose of improving performance (usually by reducing the number of joins). Should not be used for relations that change often or in cases where disk space is scarce.

**Dense index**: An index in which the underlying data structure has a pointer to each record among the data pages. Clustering indexes can be dense in some systems (e.g., ORACLE). Nonclustering indexes are always dense.

**E-commerce applications**: Applications entailing access to a website and a back end database system.

**Hash structure**: A tree structure whose root is a function, called the hash function. Given a key, the hash function returns a page that contains pointers to records holding that key or is the root of an overflow chain. Should be used when selective equality queries and updates are the dominant access patterns.

**Heap**: In the absence of a clustering index, the tuples of a table will be laid out in their order of insertion. Such a layout is called a heap. (Some systems, such as RDB, reuse the space in the interior of heaps, but most do not.)

**Hit ratio**: The number of logical accesses satisfied by the database buffer divided by the total number of logical accesses.

**Index**: A data organization to speed the execution of queries on tables or object-oriented collections. It consists of a data structure, e.g., a B-tree or hash structure, and a table organization.

**Index organized table**: A table clustered based on its primary key or on a row id if no primary key is defined.

**Locking**: The activity of obtaining and releasing read locks and write locks (see corresponding entries) for the purposes of concurrent synchronization (concurrency control) among transactions.

**Nonclustering index**: A dense index that puts no constraints on the table organization, also known as a secondary index. For contrast, see **clustering index**.

**Normalized**: A relation $R$ is normalized if every functional dependency “$X$ functionally determines $A$,” where $A$ and the attributes in $X$ are contained in $R$ (but $A$ does not belong to $X$), has the property that $X$ is the key or a superset of the key of $R$. $X$ functionally determines $A$ if any two tuples with the same $X$ values have the same $A$ value. $X$ is a key if no two records have the same values on all attributes of $X$.

**On-line transaction processing**: The class of applications where the transactions are short, typically 10 disk I/Os or fewer per transaction, the queries are simple, typically point and multipoint queries, and the frequency of updates is high.

**Read lock**: If a transaction $T$ holds a read lock on a data item $x$, then no other transaction can obtain a write lock on $x$. 
Seek: Moving the read/write head of a disk to the proper track.

Serializability: The assurance that each transaction in a database system will appear to execute in isolation of all others. Equivalently, the assurance that a concurrent execution of committed transactions will appear to execute in serial order as far as their input/output behaviors are concerned.

Solid State Drive (SSD): A storage device that is composed of tens of flash chips connected to a microcontroller. The current generation of SSD exposes a block device interface similar to the magnetic hard drive interface. The micro-controller runs a firmware called flash translation layer (or FTL) that maps operations on logical block addresses onto operations on physical blocks on flash. SSDs can be directly installed on a server’s memory bus (PCI bus), or they can be accessed as IO devices (via a SATA interface).

Sparse index: An index in which the underlying data structure contains exactly one pointer to each data page. Only clustering indexes can be sparse.

Track: A narrow ring on a single platter of a disk. If the disk head over a platter does not move, then a track will pass under that head in one rotation. The implication is that reading or writing a track does not take much more time than reading or writing a portion of a track.

Transaction: A program fragment delimited by Commit statements having database accesses that are supposed to appear as if they execute alone on the database. A typical transaction may process a purchase by increasing inventory and decreasing cash.

Two-phase locking: An algorithm for concurrency control whereby a transaction acquires a write lock on $x$ before writing $x$ and holds that lock until after its last write of $x$; acquires a read or write lock on $x$ before reading $x$ and holds that lock until after its last read of $x$; and never releases a lock on any item $x$ before obtaining a lock on any (perhaps different) item $y$. Two-phase locking can encounter deadlock. The database system resolves this by rolling back one of the transactions involved in the deadlock.

Vertical partitioning: A method of dividing each record (or object) of a table (or collection of objects) so that some attributes, including a key, of the record (or object) are in one location and others are in another location, possibly another disk. For example, the account id and the current balance may be in one location and the account id and the address information of each tuple may be in another location.

Write lock: If a transaction $T$ holds a write lock on a datum $x$, then no other transaction can obtain any lock on $x$.

References