There was a great debate at the annual ACM SIGFIDET (now SIGMOD) meeting in Ann Arbor, Michigan, in 1974 between Ted Codd, the creator of the relational database model, and Charlie Bachman, the technical creative mind behind the network database model and the subsequent CODASYL report. The debate centered on which logical model was the best database model, and it had continued on in the academic journals and trade magazines for almost 30 more years until Codd’s death in 2003. Since that original debate, many database systems have been built to support each of these models, and although the relational model eventually dominated the database industry, the underlying physical database structures used by both types of systems were actually evolving in sync. Originally the main decision for physical design was the type of indexing the system was able to do, with B+tree indexing eventually dominating the scene for almost all systems. Later, other concepts like clustering and partitioning became important, but these methods were becoming less and less related to the logical structures being debated in the 1970s.

Logical database design, that is, the design of basic data relationships and their definition in a particular database system, is largely the domain of application designers and programmers. The work of these designers can effectively be done with tools, such as ERwin Data Modeller or Rational Rose with UML, as well as with a purely manual approach. Physical database design, the creation of efficient data storage, and retrieval
mechanisms on the computing platform you are using are typically the domain of the
database administrator (DBA), who has a variety of vendor-supplied tools available
today to help design the most efficient databases. This book is devoted to the physical
design methodologies and tools most popular for relational databases today. We use
examples from the most common systems—Oracle, DB2 (IBM), and SQL Server
(Microsoft)—to illustrate the basic concepts.

1.1 Motivation—The Growth of Data and Increasing
Relevance of Physical Database Design

Does physical database design really matter? Absolutely. Some computing professionals
currently run their own consulting businesses doing little else than helping customers
improve their table indexing design. Impressive as this is, what is equally astounding are
claims about improving the performance of problem queries by as much as 50 times.
Physical database design is really motivated by data volume. After all, a database with a
few rows of data really has no issues with physical database design, and the performance
of applications that access a tiny database cannot be deeply affected by the physical
design of the underlying system. In practical terms, index selection really does not mat-
ter much for a database with 20 rows of data. However, as data volumes rise, the physi-
cal structures that underlie its access patterns become increasingly critical.

A number of factors are spurring the dramatic growth of data in all three of its cap-
tured forms: structured (relational tuples), semistructured (e.g., XML), and unstruc-
tured data (e.g., audio/video). Much of the growth can be attributed to the rapid expan-
sion and ubiquitous use of networked computers and terminals in every home, business,
and store in the industrialized world. The data volumes are now taking a further leap
forward with the rapid adoption of personal communication devices like cell phones
and PDAs, which are also networked and used to share data. Databases measured in the
tens of terabytes have now become commonplace in enterprise systems. Following the
mapping of the human genome’s three billion chemical base pairs, pharmaceutical com-
panies are now exploring genetic engineering research based on the networks of proteins
that overlay the human genomes, resulting in data analysis on databases several
petabytes in size (a petabyte is one thousand terabytes, or one million gigabytes). Table
1.1 shows data from a 1999 survey performed by the University of California at Berke-
ley. You can see in this study that the data stored on magnetic disk is growing at a rate of
100% per year for departmental and enterprise servers. In fact nobody is sure exactly
where the growth patterns will end, or if they ever will.

There’s something else special that has happened that’s driving up the data volumes.
It happened so quietly that seemingly nobody bothered to mention it, but the change is
quantitative and profound. Around the year 2000 the price of storage dropped to a
point where it became cheaper to store data on computer disks than on paper (Figure
1.1 Motivation—The Growth of Data and Increasing Relevance of Physical Database Design

In fact, this probably was a great turning point in the history of the development of western civilization. For over 2,000 years civilization has stored data in written text—on parchment, papyrus, or paper. Suddenly and quietly that paradigm has begun to sunset. Now the digitization of text is not only of interest for sharing and analysis, but it is also more economical.

The dramatic growth patterns change the amount of data that relational database systems must access and manipulate, but they do not change the speed at which operations must complete. In fact, to a large degree, the execution goals for data processing systems are defined more by human qualities than by computers: the time a person is willing to wait for a transaction to complete while standing at an automated banking machine or the number of available off-peak hours between closing time of a business in the evening and the resumption of business in the morning. These are constraints that are defined largely by what humans expect and they are quite independent of the data volumes being operated on. While data volumes and analytic complexity are growing.

Table 1.1 Worldwide Production of Original Content, Stored Digitally, in Terabytes*

<table>
<thead>
<tr>
<th>Storage Medium</th>
<th>Type of Content</th>
<th>Terabytes/Year, Upper Estimate</th>
<th>Terabytes/Year, Lower Estimate</th>
<th>Growth Rate, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper</td>
<td>Books</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Newspapers</td>
<td>25</td>
<td>2</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td>Periodicals</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Office documents</td>
<td>195</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal:</strong></td>
<td><strong>240</strong></td>
<td><strong>23</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td>Film</td>
<td>Photographs</td>
<td>410,000</td>
<td>41,000</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Cinema</td>
<td>16</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>X-Rays</td>
<td>17,200</td>
<td>17,200</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal:</strong></td>
<td><strong>427,216</strong></td>
<td><strong>58,216</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td>Optical</td>
<td>Music CDs</td>
<td>58</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Data CDs</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>DVDs</td>
<td>22</td>
<td>22</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal:</strong></td>
<td><strong>83</strong></td>
<td><strong>31</strong></td>
<td><strong>70</strong></td>
</tr>
<tr>
<td>Magnetic</td>
<td>Camcorder Tape</td>
<td>300,000</td>
<td>300,000</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>PC Disk Driver</td>
<td>766,000</td>
<td>7,660</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Departmental Servers</td>
<td>460,000</td>
<td>161,000</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Enterprise Servers</td>
<td>167,000</td>
<td>108,550</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal:</strong></td>
<td><strong>1,693,000</strong></td>
<td><strong>577,210</strong></td>
<td><strong>55</strong></td>
</tr>
<tr>
<td><strong>TOTAL:</strong></td>
<td></td>
<td><strong>2,120,539</strong></td>
<td><strong>635,480</strong></td>
<td><strong>50</strong></td>
</tr>
</tbody>
</table>

*Source: University of California at Berkeley study, 1999.*
rapidly, our expectations as humans are changing at a much slower rate. Some relief is found in the increasing power of modern data servers because as the data volumes grow, the computing power behind them is increasing as well. However, the phenomenon of increasing processing power is mitigated by the need to consolidate server technology to reduce IT expenses, so as a result, as servers grow in processing power they are often used for an increasing number of purposes rather than being used to perform a single purpose faster.

Although CPU power has been improving following Moore’s Law, doubling roughly every 18 months since the mid 1970s, disk speeds have been increasing at a more modest pace (see Chapter 13 for a more in-depth discussion of Moore’s Law). Finally, data is increasingly being used to detect “information” not just process “data,” and the rise of on-line analytical processing (OLAP) and data mining and other forms
of business intelligence computing has led to a dramatic increase in the complexity of the processing that is required. These factors motivate the need for complex and sophisticated approaches to physical database design. Why? By exploiting design techniques a practitioner can reduce the processing time for operations in some cases by several orders of magnitude. Improving computational efficiency by a thousand times is real, and valuable; and when you’re waiting at the bank machine to get your money, or waiting for an analysis of business trading that will influence a multimillion dollar investment decision, it’s downright necessary.

1.2 Database Life Cycle

The database life cycle incorporates the basic steps involved in designing a logical database from conceptual modeling of user requirements through database management system (DBMS) specific table definitions, and a physical database that is indexed, partitioned, clustered, and selectively materialized to maximize real performance. For a distributed database, physical design also involves allocating data across a computer network. Once the design is completed, the life cycle continues with database implementation and maintenance. The database life cycle is shown in Figure 1.2. Physical database design (step 3 below) is defined in the context of the entire database life cycle to show its relationship to the other design steps.

1. Requirements analysis. The database requirements are determined by interviewing both the producers and users of data and producing a formal requirements specification. That specification includes the data required for processing, the natural data relationships, and the software platform for the database implementation.

2. Logical database design. Logical database design develops a conceptual model of the database from a set of user requirements and refines that model into normalized SQL tables. The goal of logical design is to capture the reality of the user’s world in terms of data elements and their relationships so that queries and updates to that data can be programmed easily. The global schema, a conceptual data model diagram that shows all the data and their relationships, is developed using techniques such as entity-relationship (ER) modeling or the Unified Modeling Language (UML). The data model constructs must ultimately be integrated into a single global schema and then transformed into normalized SQL tables. Normalized tables (particularly third normal form or 3NF) are tables that are decomposed or split into smaller tables to eliminate loss of data integrity due to certain delete commands.

We note here that some database tool vendors use the term logical model to refer to the conceptual data model, and they use the term physical model to refer
Figure 1.2  Database life cycle.
to the DBMS-specific implementation model (e.g., SQL tables). We also note that many conceptual data models are obtained not from scratch, but from the process of reverse engineering from an existing DBMS-specific schema [Silberschatz 2006]. Our definition of the physical model is given below.

3. **Physical database design.** The physical database design step involves the selection of indexes, partitioning, clustering, and selective materialization of data. Physical database design (as treated in this book) begins after the SQL tables have been defined and normalized. It focuses on the methods of storing and accessing those tables on disk that enable the database to operate with high efficiency. The goal of physical design is to maximize the performance of the database across the entire spectrum of applications written on it. The physical resources that involve time delays in executing database applications include the CPU, I/O (e.g., disks), and computer networks. Performance is measured by the time delays to answer a query or complete an update for an individual application, and also by the throughput (in transactions per second) for the entire database system over the full set of applications in a specified unit of time.

4. **Database implementation, monitoring, and modification.** Once the logical and physical design is completed, the database can be created through implementation of the formal schema using the data definition language (DDL) of a DBMS. Then the data manipulation language (DML) can be used to query and update the database, as well as to set up indexes and establish constraints such as referential integrity. The language SQL contains both DDL and DML constructs; for example, the “create table” command represents DDL, and the “select” command represents DML.

As the database begins operation, monitoring indicates whether performance requirements are being met. If they are not being satisfied, modifications should be made to improve performance. Other modifications may be necessary when requirements change or end-user expectations increase with good performance. Thus, the life cycle continues with monitoring, redesign, and modifications.

### 1.3 Elements of Physical Design: Indexing, Partitioning, and Clustering

The physical design of a database starts with the schema definition of logical records produced by the logical design phase. A *logical record* (or record) is a named collection of data items or attributes treated as a unit by an application program. In storage, a record includes the pointers and record overhead needed for identification and processing by the database management system. A *file* is typically a set of similarly constructed records of one type, and relational tables are typically stored as files. A *physical database* is a col-
lection of interrelated records of different types, possibly including a collection of inter-related files. Query and update transactions to a database are made efficient by the implementation of certain search methods as part of the database management system.

1.3.1 Indexes

An index is a data organization set up to speed up the retrieval (query) of data from tables. In database management systems, indexes can be specified by database application programmers using the following SQL commands:

```
CREATE UNIQUE INDEX supplierNum ON supplier(snum);
/*unique index on a key*/
```

A unique index is a data structure (table) whose entries (records) consist of attribute value, pointer pairs such that each pointer contains the block address of an actual database record that has the associated attribute value as an index key value. This is known as an ordered index because the attribute (key) values in the index are ordered as ASCII values. If all the key values are letters, then the ordering is strictly alphabetical. Ordered indexes are typically stored as B+trees so that the search for the matching key value is fast. Once the key value and corresponding data block pointer are found, there is one more step to access the block containing the record you want, and a quick search in memory of that block to find the record.

Sometimes data is better accessed by an attribute other than a key, an attribute that typically has the same value appear in many records. In a unique index based on a key, the key has a unique value in every record. For a nonunique attribute, an index must have multiple attribute, pointer pairs for the same attribute value, and each pointer has the block address of a record that has one of those attribute values. In the B+tree index, the leaf nodes contain these attribute, pointer pairs that must be searched to find the records that match the attribute value. The SQL command for this kind of index, also called a secondary index, is:

```
CREATE INDEX shippingDate ON shipment (shipdate);
/*secondary index on non-key*/
```

In a variation of the secondary or nonunique index, it is possible to set up a collection of attribute values that you want to use to query a table. Each entry in the index consists of a set of attribute values and a block pointer to the record that contains exact matches for all those attribute values in the set. An example of an SQL command to set up this kind of index is:

```
CREATE INDEX shipPart ON shipment (pnum, shipdate);
/*secondary concatenated index*/
```
This kind of index is extremely efficient for queries involving both a part number (pnum) and shipping date (shipdate). For queries involving just one of these attributes, it is less efficient because of its greater size and therefore longer search time.

When we want to improve the query time for a table of data, say for instance the table we access via the nonunique index on ship dates, we could organize the database so that equivalent ship dates are stored near each other (on disk), and ship dates that are close to each other in value are stored near each other. This type index is called a clustered index. Otherwise the index is known as a nonclustered index. There can only be one clustered index per table because the physical organization of the table must be fixed.

When the physical database table is unordered, it can be organized for efficient access using a hash table index, often simply known as a hash index. This type of index is most frequently based on a key that has unique values in the data records. The attribute (key) value is passed through a function that maps it to a starting block address, known as a bucket address. The table must be set up by inserting all the records according to the hash function, and then using the same hash function to query the records later.

Another variation of indexing, a bitmap index, is commonly used for secondary indexing with multiple attribute values, and for very large databases in data warehouses. A bitmap index consists of a collection of bit vectors, with each bit vector corresponding to a particular attribute value, and for each record in the table, the bit vector is a “1” if that record has the designated bit vector value, and “0” if it does not. This is particularly useful if an attribute is sparse, that is, it has very few possible values, like gender or course grade. It would not work well for attributes like last name, job title, age, and so on. Bit vectors can be stored and accessed very efficiently, especially if they are small enough to be located in memory.

The analysis and design of indexes are discussed in detail in Chapters 2 and 4.

1.3.2 Materialized Views

When one or more tables are queried, the result can be stored in what is called a materialized view. Normally, views in SQL are stored as definitions or templates, but materialized views are stored as tables in the database just like any other table. In data warehouses, materialized views are maintained as aggregates of data in the base tables. These kinds of views are very useful to speed up queries of data that have been asked before (and frequently), or queries based on aggregates of data that can build on materialized views to answer the question instead of having to go back to the original data each time. Potentially a great deal of query time savings can be realized if the proper set of materialized views is stored. It is usually impossible to store all possible views because of storage space limitations, so some means must be found to focus on the best set of views to materialize. There is also a problem with updates—when base tables are updated, this cascades into the materialized views, which are derived from the base tables. The problem of multiple updates makes the use of materialized views less efficient, and this must
be taken into account in their design and usage. This is discussed in more detail in Chapter 5.

1.3.3 Partitioning and Multidimensional Clustering

Partitioning in physical database design is a method for reducing the workload on any one hardware component, like an individual disk, by partitioning (dividing) the data over several disks. This has the effect of balancing the workload across the system and preventing bottlenecks. In range partitioning, the data attribute values are sorted and ranges of values are selected so that each range has roughly the same number of records. Records in a given range are allocated to a specific disk so it can be processed independently of other ranges. The details of partitioning across disks are discussed in Chapter 7.

Multidimensional clustering (MDC) is a technique by which data can be clustered by dimensions, such as location, timeframe, or product type. In particular, MDC allows data to be clustered by many dimensions at the same time, such as ice skates sold in Wisconsin during the month of December. The clusters are meant to take advantage of known and anticipated workloads on this data. MDC is developed in detail in Chapter 8.

1.3.4 Other Methods for Physical Database Design

There are many other ways to make data access more efficient in a database. For instance, data compression is a technique that allows more data to fit into a fixed amount of space (on disk) and therefore accessed faster if data needs to be scanned a lot. The overhead for compression is in the algorithm to transform the original data into the compressed form for storage, and then to transform the compressed form back to the original for display purposes.

Data striping, or just striping, is a technique for distributing data that needs to be accessed together across multiple disks to achieve a greater degree of parallelism and load balancing, both of which makes system throughput increase and generally lowers query times. This is particularly suited to disk array architectures like RAID (redundant arrays of independent disks) where data can be accessed in parallel across multiple disks in an organized way.

Another way to improve database reliability includes data redundancy techniques like mirroring, in which data is duplicated on multiple disks. The downside of redundancy is having to update multiple copies of data each time a change is required in the database, as well as the extra storage space required. Storage space is getting cheaper every day, but time is not. On the other hand, data that is never or infrequently updated may lend itself nicely to be stored redundantly.
As part of the physical design, the global schema can sometimes be refined in limited ways to reflect processing (query and transaction) requirements if there are obvious large gains to be made in efficiency. This is called denormalization. It consists of selecting dominant processes on the basis of high frequency, high volume, or explicit priority; defining simple extensions to tables that will improve query performance; evaluating total cost for query, update, and storage; and considering the side effects, such as possible loss of integrity. Details are given in Chapter 15.

1.4 Why Physical Design Is Hard

Physical database design involves dozens and often hundreds of variables, which are difficult to keep track of, especially when their effects are very often interrelated to the various design solutions proposed. The individual computations of performance based on a given index mechanism or partition algorithm may take several hours by hand, and performance analysis is often based on the comparison of many different configurations and load conditions, thus requiring thousands of computations. This has given rise to automated tools such as IBM’s DB2 Design Advisor, Oracle’s SQL Access Advisor, Oracle’s SQL Tuning Advisor, and Microsoft’s Database Tuning Advisor (DTA), formerly known as the Index Tuning Wizard. These tools make database tuning and performance analysis manageable, allowing the analyst to focus on solutions and tradeoffs while taking care of the myriad of computations that are needed. We will look at both manual analysis and automatic design tools for physical database design in this book.

TIPS AND INSIGHTS FOR DATABASE PROFESSIONALS

• Tip 1. The truth is out there, but you may not need it. Every database has a theoretically perfect, or "optimal", physical design. In reality almost nobody ever finds it because the search complexity is too high and the validation process too cumbersome. Database design is really hard problem. However, the complexity is mitigated by the practical fact that at the end of the day what matters most is not whether the database performance is as good as it can theoretically be, but whether the applications that use the database perform "good enough" so that their users are satisfied. Good enough is a vague and subjective definition of course. In most cases, while the perfect database design is usually elusive, one that performs more than 85% of optimal can be achieved by mere mortals.
Tip 2. Be prepared to tinker. The possibilities are endless, and you will never be able to explore them all. But with some wisdom and insight you and some playing around with possibilities you can go far. Trial and error is part of the process.

Tip 3. Use the tools at your disposal. Throughout this book we will describe various techniques and methods for physical database design. Many database design perform an order of magnitude worse than they could simply because the designer didn’t bother to use the techniques available. Database designs does not begin and end with simple single column index selection. By exploiting features like memory tuning, materialized views, range partitioning, multidimensional clustering, clustering indexes, or shared nothing partitioning you can dramatically improve on a basic database design, especially for complex query processing.

1.5 Literature Summary

Database system and design textbooks and practitioners’ guides that give serious attention to the principles of physical database design include Burleson [2005], Elmasri and Navathe [2003], Garcia-Molina, Ullman, and Widom [2000, 2001], Ramakrishnan and Gehrke [2004], Shasha and Bonnet [2003], and Silberschatz, Korth, and Sudarshan [2006].

Knowledge of logical data modeling and physical database design techniques is important for database practitioners and application developers. The database life cycle shows what steps are needed in a methodical approach to database design from logical design, which is independent of the system environment, to physical design, which is based on maximizing the performance of the database under various workloads.


Oracle—SQL Tuning Advisor, at http://www.oracle-base.com/articles/10g/AutomaticSQLTuning10g.php.


