Foundations and Trends® in
Databases
Vol. 1, No. 2 (2007) 141–259
© 2007 J. M. Hellerstein, M. Stonebraker
and J. Hamilton
DOI: 10.1561/1900000002

now
the essence of knowledge

Architecture of a Database System

Joseph M. Hellerstein[1], Michael Stonebraker[2] and James Hamilton[3]

[1] *University of California, Berkeley, USA, hellerstein@cs.berkeley.edu*
[2] *Massachusetts Institute of Technology, USA*
[3] *Microsoft Research, USA*

Abstract

Database Management Systems (DBMSs) are a ubiquitous and critical component of modern computing, and the result of decades of research and development in both academia and industry. Historically, DBMSs were among the earliest multi-user server systems to be developed, and thus pioneered many systems design techniques for scalability and reliability now in use in many other contexts. While many of the algorithms and abstractions used by a DBMS are textbook material, there has been relatively sparse coverage in the literature of the systems design issues that make a DBMS work. This paper presents an architectural discussion of DBMS design principles, including process models, parallel architecture, storage system design, transaction system implementation, query processor and optimizer architectures, and typical shared components and utilities. Successful commercial and open-source systems are used as points of reference, particularly when multiple alternative designs have been adopted by different groups.

1

Introduction

Database Management Systems (DBMSs) are complex, mission-critical software systems. Today's DBMSs embody decades of academic and industrial research and intense corporate software development. Database systems were among the earliest widely deployed online server systems and, as such, have pioneered design solutions spanning not only data management, but also applications, operating systems, and networked services. The early DBMSs are among the most influential software systems in computer science, and the ideas and implementation issues pioneered for DBMSs are widely copied and reinvented.

For a number of reasons, the lessons of database systems architecture are not as broadly known as they should be. First, the applied database systems community is fairly small. Since market forces only support a few competitors at the high end, only a handful of successful DBMS implementations exist. The community of people involved in designing and implementing database systems is tight: many attended the same schools, worked on the same influential research projects, and collaborated on the same commercial products. Second, academic treatment of database systems often ignores architectural issues. Textbook presentations of database systems traditionally focus on algorithmic

and theoretical issues — which are natural to teach, study, and test — without a holistic discussion of system architecture in full implementations. In sum, much conventional wisdom about how to build database systems is available, but little of it has been written down or communicated broadly.

In this paper, we attempt to capture the main architectural aspects of modern database systems, with a discussion of advanced topics. Some of these appear in the literature, and we provide references where appropriate. Other issues are buried in product manuals, and some are simply part of the oral tradition of the community. Where applicable, we use commercial and open-source systems as examples of the various architectural forms discussed. Space prevents, however, the enumeration of the exceptions and finer nuances that have found their way into these multi-million line code bases, most of which are well over a decade old. Our goal here is to focus on overall system design and stress issues not typically discussed in textbooks, providing useful context for more widely known algorithms and concepts. We assume that the reader is familiar with textbook database systems material (e.g., [72] or [83]) and with the basic facilities of modern operating systems such as UNIX, Linux, or Windows. After introducing the high-level architecture of a DBMS in the next section, we provide a number of references to background reading on each of the components in Section 1.2.

1.1 Relational Systems: The Life of a Query

The most mature and widely used database systems in production today are relational database management systems (RDBMSs). These systems can be found at the core of much of the world's application infrastructure including e-commerce, medical records, billing, human resources, payroll, customer relationship management and supply chain management, to name a few. The advent of web-based commerce and community-oriented sites has only increased the volume and breadth of their use. Relational systems serve as the repositories of record behind nearly all online transactions and most online content management systems (blogs, wikis, social networks, and the like). In addition to being important software infrastructure, relational database systems serve as

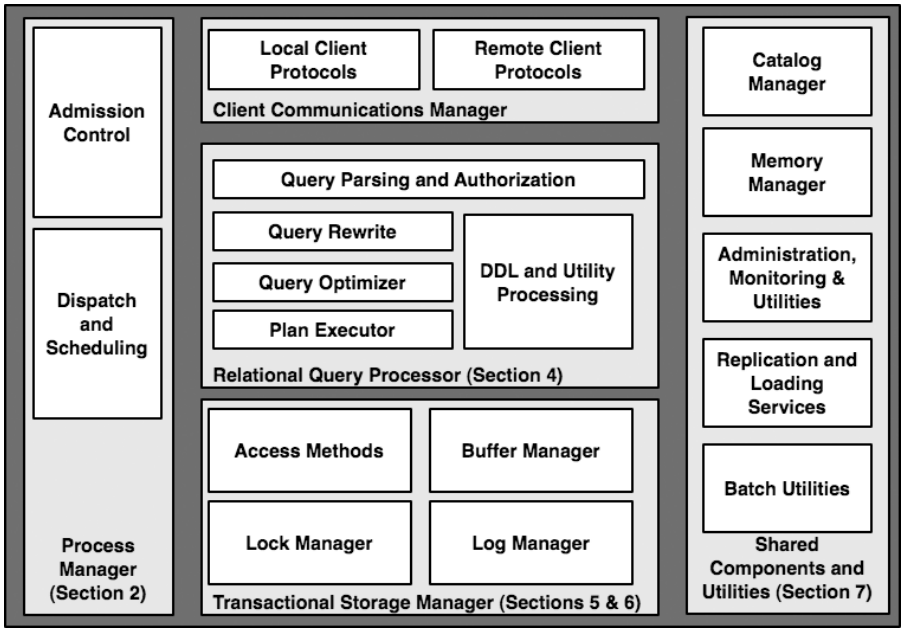


Fig. 1.1 Main components of a DBMS.

a well-understood point of reference for new extensions and revolutions in database systems that may arise in the future. As a result, we focus on relational database systems throughout this paper.

At heart, a typical RDBMS has five main components, as illustrated in Figure 1.1. As an introduction to each of these components and the way they fit together, we step through the life of a query in a database system. This also serves as an overview of the remaining sections of the paper.

Consider a simple but typical database interaction at an airport, in which a gate agent clicks on a form to request the passenger list for a flight. This button click results in a single-query transaction that works roughly as follows:

1. The personal computer at the airport gate (the "client") calls an API that in turn communicates over a network to establish a connection with the *Client Communications Manager* of a DBMS (top of Figure 1.1). In some cases, this connection

is established between the client and the database server directly, e.g., via the ODBC or JDBC connectivity protocol. This arrangement is termed a "two-tier" or "client-server" system. In other cases, the client may communicate with a "middle-tier server" (a web server, transaction processing monitor, or the like), which in turn uses a protocol to proxy the communication between the client and the DBMS. This is usually called a "three-tier" system. In many web-based scenarios there is yet another "application server" tier between the web server and the DBMS, resulting in four tiers. Given these various options, a typical DBMS needs to be compatible with many different connectivity protocols used by various client drivers and middleware systems. At base, however, the responsibility of the DBMS' client communications manager in all these protocols is roughly the same: to establish and remember the connection state for the caller (be it a client or a middleware server), to respond to SQL commands from the caller, and to return both data and control messages (result codes, errors, etc.) as appropriate. In our simple example, the communications manager would establish the security credentials of the client, set up state to remember the details of the new connection and the current SQL command across calls, and forward the client's first request deeper into the DBMS to be processed.

2. Upon receiving the client's first SQL command, the DBMS must assign a "thread of computation" to the command. It must also make sure that the thread's data and control outputs are connected via the communications manager to the client. These tasks are the job of the DBMS *Process Manager* (left side of Figure 1.1). The most important decision that the DBMS needs to make at this stage in the query regards *admission control*: whether the system should begin processing the query immediately, or defer execution until a time when enough system resources are available to devote to this query. We discuss Process Management in detail in Section 2.

3. Once admitted and allocated as a thread of control, the gate agent's query can begin to execute. It does so by invoking the code in the *Relational Query Processor* (center, Figure 1.1). This set of modules checks that the user is authorized to run the query, and compiles the user's SQL query text into an internal *query plan*. Once compiled, the resulting query plan is handled via the plan executor. The plan executor consists of a suite of "operators" (relational algorithm implementations) for executing any query. Typical operators implement relational query processing tasks including joins, selection, projection, aggregation, sorting and so on, as well as calls to request data records from lower layers of the system. In our example query, a small subset of these operators — as assembled by the query optimization process — is invoked to satisfy the gate agent's query. We discuss the query processor in Section 4.
4. At the base of the gate agent's query plan, one or more operators exist to request data from the database. These operators make calls to fetch data from the DBMS' *Transactional Storage Manager* (Figure 1.1, bottom), which manages all data access (read) and manipulation (create, update, delete) calls. The storage system includes algorithms and data structures for organizing and accessing data on disk ("access methods"), including basic structures like tables and indexes. It also includes a buffer management module that decides when and what data to transfer between disk and memory buffers. Returning to our example, in the course of accessing data in the access methods, the gate agent's query must invoke the transaction management code to ensure the well-known "ACID" properties of transactions [30] (discussed in more detail in Section 5.1). Before accessing data, locks are acquired from a lock manager to ensure correct execution in the face of other concurrent queries. If the gate agent's query involved updates to the database, it would interact with the log manager to ensure that the transaction was durable if committed, and fully undone if aborted.

In Section 5, we discuss storage and buffer management in more detail; Section 6 covers the transactional consistency architecture.

5. At this point in the example query's life, it has begun to access data records, and is ready to use them to compute results for the client. This is done by "unwinding the stack" of activities we described up to this point. The access methods return control to the query executor's operators, which orchestrate the computation of result tuples from database data; as result tuples are generated, they are placed in a buffer for the client communications manager, which ships the results back to the caller. For large result sets, the client typically will make additional calls to fetch more data incrementally from the query, resulting in multiple iterations through the communications manager, query executor, and storage manager. In our simple example, at the end of the query the transaction is completed and the connection closed; this results in the transaction manager cleaning up state for the transaction, the process manager freeing any control structures for the query, and the communications manager cleaning up communication state for the connection.

Our discussion of this example query touches on many of the key components in an RDBMS, but not all of them. The right-hand side of Figure 1.1 depicts a number of shared components and utilities that are vital to the operation of a full-function DBMS. The catalog and memory managers are invoked as utilities during any transaction, including our example query. The catalog is used by the query processor during authentication, parsing, and query optimization. The memory manager is used throughout the DBMS whenever memory needs to be dynamically allocated or deallocated. The remaining modules listed in the rightmost box of Figure 1.1 are utilities that run independently of any particular query, keeping the database as a whole well-tuned and reliable. We discuss these shared components and utilities in Section 7.

1.2 Scope and Overview

In most of this paper, our focus is on architectural fundamentals supporting core database functionality. We do not attempt to provide a comprehensive review of database algorithmics that have been extensively documented in the literature. We also provide only minimal discussion of many extensions present in modern DBMSs, most of which provide features beyond core data management but do not significantly alter the system architecture. However, within the various sections of this paper we note topics of interest that are beyond the scope of the paper, and where possible we provide pointers to additional reading.

We begin our discussion with an investigation of the overall architecture of database systems. The first topic in any server system architecture is its overall process structure, and we explore a variety of viable alternatives on this front, first for uniprocessor machines and then for the variety of parallel architectures available today. This discussion of core server system architecture is applicable to a variety of systems, but was to a large degree pioneered in DBMS design. Following this, we begin on the more domain-specific components of a DBMS. We start with a single query's view of the system, focusing on the relational query processor. Following that, we move into the storage architecture and transactional storage management design. Finally, we present some of the shared components and utilities that exist in most DBMSs, but are rarely discussed in textbooks.

2

Process Models

When designing any multi-user server, early decisions need to be made regarding the execution of concurrent user requests and how these are mapped to operating system processes or threads. These decisions have a profound influence on the software architecture of the system, and on its performance, scalability, and portability across operating systems.[1] In this section, we survey a number of options for DBMS process models, which serve as a template for many other highly concurrent server systems. We begin with a simplified framework, assuming the availability of good operating system support for threads, and we initially target only a uniprocessor system. We then expand on this simplified discussion to deal with the realities of how modern DBMSs implement their process models. In Section 3, we discuss techniques to exploit clusters of computers, as well as multi-processor and multi-core systems.

The discussion that follows relies on these definitions:

- An *Operating System Process* combines an operating system (OS) program execution unit (a thread of control) with an

[1] Many but not all DBMSs are designed to be portable across a wide variety of host operating systems. Notable examples of OS-specific DBMSs are DB2 for zSeries and Microsoft SQL Server. Rather than using only widely available OS facilities, these products are free to exploit the unique facilities of their single host.

address space private to the process. Included in the state maintained for a process are OS resource handles and the security context. This single unit of program execution is scheduled by the OS kernel and each process has its own unique address space.

- An *Operating System Thread* is an OS program execution unit without additional private OS context and without a private address space. Each OS thread has full access to the memory of other threads executing within the same *multi-threaded* OS Process. Thread execution is scheduled by the operating system kernel scheduler and these threads are often called "kernel threads" or k-threads.
- A *Lightweight Thread Package* is an application-level construct that supports multiple threads within a single OS process. Unlike OS threads scheduled by the OS, lightweight threads are scheduled by an application-level thread scheduler. The difference between a lightweight thread and a kernel thread is that a lightweight thread is scheduled in user-space without kernel scheduler involvement or knowledge. The combination of the user-space scheduler and all of its lightweight threads run within a single OS process and appears to the OS scheduler as a single thread of execution.

  Lightweight threads have the advantage of faster thread switches when compared to OS threads since there is no need to do an OS kernel mode switch to schedule the next thread. Lightweight threads have the disadvantage, however, that any blocking operation such as a synchronous I/O by any thread will block all threads in the process. This prevents any of the other threads from making progress while one thread is blocked waiting for an OS resource. Lightweight thread packages avoid this by (1) issuing only asynchronous (non-blocking) I/O requests and (2) not invoking any OS operations that could block. Generally, lightweight threads offer a more difficult programming model than writing software based on either OS processes or OS threads.

- Some DBMSs implement their own lightweight thread (LWT) packages. These are a special case of general LWT packages. We refer to these threads as *DBMS threads* and simply *threads* when the distinction between DBMS, general LWT, and OS threads are unimportant to the discussion.
- A *DBMS Client* is the software component that implements the API used by application programs to communicate with a DBMS. Some example database access APIs are JDBC, ODBC, and OLE/DB. In addition, there are a wide variety of proprietary database access API sets. Some programs are written using embedded SQL, a technique of mixing programming language statements with database access statements. This was first delivered in IBM COBOL and PL/I and, much later, in SQL/J which implements embedded SQL for Java. Embedded SQL is processed by preprocessors that translate the embedded SQL statements into direct calls to data access APIs. Whatever the syntax used in the client program, the end result is a sequence of calls to the DBMS data access APIs. Calls made to these APIs are marshaled by the DBMS client component and sent to the DBMS over some communications protocol. The protocols are usually proprietary and often undocumented. In the past, there have been several efforts to standardize client-to-database communication protocols, with Open Group DRDA being perhaps the best known, but none have achieved broad adoption.
- A *DBMS Worker* is the thread of execution in the DBMS that does work on behalf of a DBMS Client. A 1:1 mapping exists between a DBMS worker and a DBMS Client: the DBMS worker handles all SQL requests from a single DBMS Client. The DBMS client sends SQL requests to the DBMS server. The worker executes each request and returns the result to the client. In what follows, we investigate the different approaches commercial DBMSs use to map DBMS workers onto OS threads or processes. When the distinction is

significant, we will refer to them as *worker threads* or *worker processes.* Otherwise, we refer to them simply as workers or DBMS workers.

2.1 Uniprocessors and Lightweight Threads

In this subsection, we outline a simplified DBMS process model taxonomy. Few leading DBMSs are architected exactly as described in this section, but the material forms the basis from which we will discuss current generation production systems in more detail. Each of the leading database systems today is, at its core, an extension or enhancement of at least one of the models presented here.

We start by making two simplifying assumptions (which we will relax in subsequent sections):

1. *OS thread support*: We assume that the OS provides us with efficient support for kernel threads and that a process can have a very large number of threads. We also assume that the memory overhead of each thread is small and that the context switches are inexpensive. This is arguably true on a number of modern OS today, but was certainly not true when most DBMSs were first designed. Because OS threads either were not available or scaled poorly on some platforms, many DBMSs are implemented without using the underlying OS thread support.
2. *Uniprocessor hardware*: We will assume that we are designing for a single machine with a single CPU. Given the ubiquity of multi-core systems, this is an unrealistic assumption even at the low end. This assumption, however, will simplify our initial discussion.

In this simplified context, a DBMS has three natural process model options. From the simplest to the most complex, these are: (1) *process per DBMS worker*, (2) *thread per DBMS worker*, and (3) *process pool.* Although these models are simplified, all three are in use by commercial DBMS systems today.

2.1.1 Process per DBMS Worker

The *process per DBMS worker* model (Figure 2.1) was used by early DBMS implementations and is still used by many commercial systems today. This model is relatively easy to implement since DBMS workers are mapped directly onto OS processes. The OS scheduler manages the timesharing of DBMS workers and the DBMS programmer can rely on OS protection facilities to isolate standard bugs like memory overruns. Moreover, various programming tools like debuggers and memory checkers are well-suited to this process model. Complicating this model are the in-memory data structures that are shared across DBMS connections, including the lock table and buffer pool (discussed in more detail in Sections 6.3 and 5.3, respectively). These shared data structures must be explicitly allocated in OS-supported shared memory accessible across all DBMS processes. This requires OS support (which is widely available) and some special DBMS coding. In practice, the

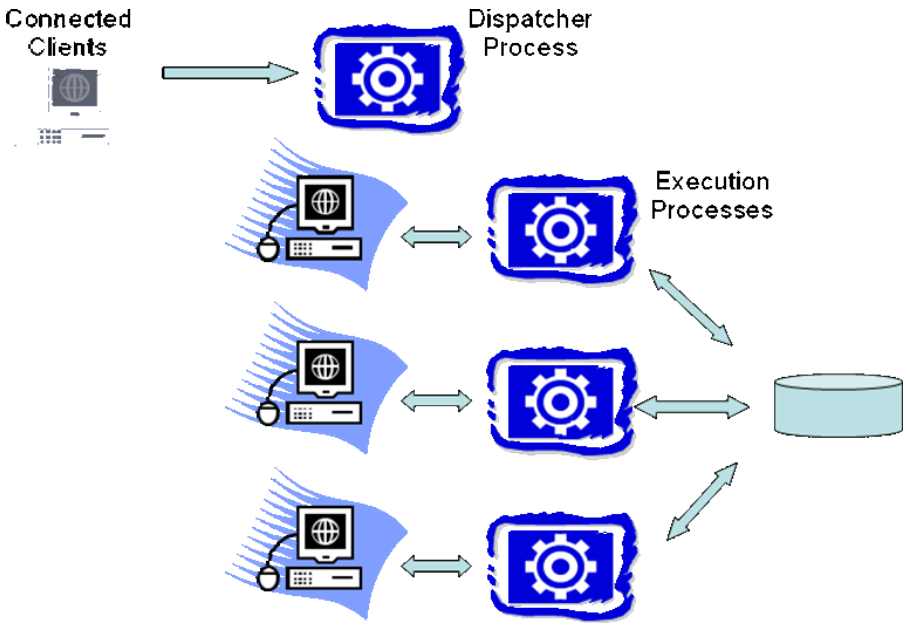


Fig. 2.1 Process per DBMS worker model: each DBMS worker is implemented as an OS process.

required extensive use of shared memory in this model reduces some of the advantages of address space separation, given that a good fraction of "interesting" memory is shared across processes.

In terms of scaling to very large numbers of concurrent connections, *process per DBMS worker* is not the most attractive process model. The scaling issues arise because a process has more state than a thread and consequently consumes more memory. A process switch requires switching security context, memory manager state, file and network handle tables, and other process context. This is not needed with a thread switch. Nonetheless, the *process per DBMS worker* model remains popular and is supported by IBM DB2, PostgreSQL, and Oracle.

2.1.2 Thread per DBMS Worker

In the *thread per DBMS worker* model (Figure 2.2), a single multithreaded process hosts all the DBMS worker activity. A dispatcher

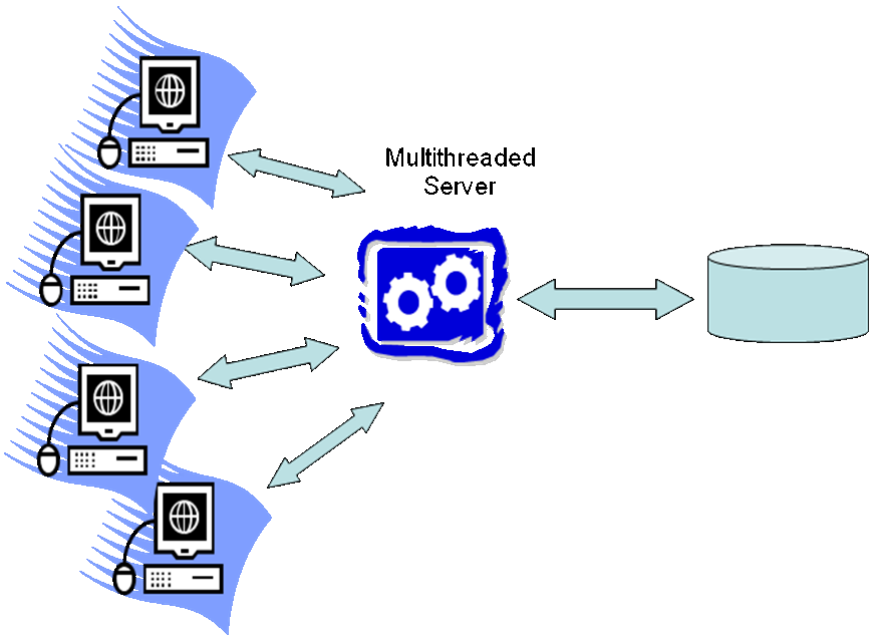


Fig. 2.2 Thread per DBMS worker model: each DBMS worker is implemented as an OS thread.

thread (or a small handful of such threads) listens for new DBMS client connections. Each connection is allocated a new thread. As each client submits SQL requests, the request is executed entirely by its corresponding thread running a DBMS worker. This thread runs within the DBMS process and, once complete, the result is returned to the client and the thread waits on the connection for the next request from that same client.

The usual multi-threaded programming challenges arise in this architecture: the OS does not protect threads from each other's memory overruns and stray pointers; debugging is tricky, especially with race conditions; and the software can be difficult to port across OS due to differences in threading interfaces and multi-threaded scaling. Many of the multi-programming challenges of the *thread per DBMS worker* model are also found in the *process per DBMS worker* model due to the extensive use of shared memory.

Although thread API differences across OSs have been minimized in recent years, subtle distinctions across platforms still cause hassles in debugging and tuning. Ignoring these implementation difficulties, the *thread per DBMS worker* model scales well to large numbers of concurrent connections and is used in some current-generation production DBMS systems, including IBM DB2, Microsoft SQL Server, MySQL, Informix, and Sybase.

2.1.3 Process Pool

This model is a variant of *process per DBMS worker*. Recall that the advantage of *process per DBMS worker* was its implementation simplicity. But the memory overhead of each connection requiring a full process is a clear disadvantage. With *process pool* (Figure 2.3), rather than allocating a full process per DBMS worker, they are hosted by a pool of processes. A central process holds all DBMS client connections and, as each SQL request comes in from a client, the request is given to one of the processes in the *process pool*. The SQL Statement is executed through to completion, the result is returned to the database client, and the process is returned to the pool to be allocated to the next request. The *process pool* size is bounded and often fixed. If a request comes in

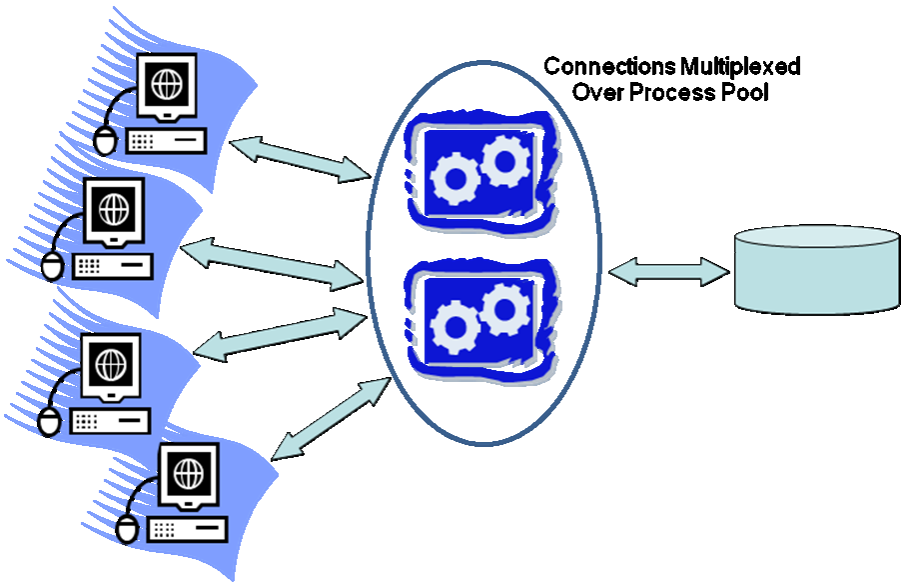


Fig. 2.3 Process Pool: each DBMS Worker is allocated to one of a pool of OS processes as work requests arrive from the Client and the process is returned to the pool once the request is processed.

and all processes are already servicing other requests, the new request must wait for a process to become available.

Process pool has all of the advantages of p*rocess per DBMS worker* but, since a much smaller number of processes are required, is considerably more memory efficient. *Process pool* is often implemented with a dynamically resizable *process pool* where the pool grows potentially to some maximum number when a large number of concurrent requests arrive. When the request load is lighter, the *process pool* can be reduced to fewer waiting processes. As with *thread per DBMS worker*, the *process pool* model is also supported by a several current generation DBMS in use today.

2.1.4 Shared Data and Process Boundaries

All models described above aim to execute concurrent client requests as independently as possible. Yet, full DBMS worker independence and isolation is not possible, since they are operating on the same shared

database. In the *thread per DBMS worker* model, data sharing is easy with all threads run in the same address space. In other models, shared memory is used for shared data structures and state. In all three models, data must be moved from the DBMS to the clients. This implies that all SQL requests need to be moved into the server processes and that all results for return to the client need to be moved back out. How is this done? The short answer is that various buffers are used. The two major types are disk I/O buffers and client communication buffers. We describe these buffers here, and briefly discuss policies for managing them.

Disk I/O buffers: The most common cross-worker data dependencies are reads and writes to the shared data store. Consequently, I/O interactions between DBMS workers are common. There are two separate disk I/O scenarios to consider: (1) database requests and (2) log requests.

- *Database I/O Requests: The Buffer Pool.* All persistent database data is staged through the DBMS *buffer pool* (Section 5.3). With *thread per DBMS worker*, the buffer pool is simply a heap-resident data structure available to all threads in the shared DBMS address space. In the other two models, the buffer pool is allocated in shared memory available to all processes. The end result in all three DBMS models is that the buffer pool is a large shared data structure available to all database threads and/or processes. When a thread needs a page to be read in from the database, it generates an I/O request specifying the disk address, and a handle to a free memory location (*frame*) in the buffer pool where the result can be placed. To flush a buffer pool page to disk, a thread generates an I/O request that includes the page's current frame in the buffer pool, and its destination address on disk. Buffer pools are discussed in more detail in Section 4.3.
- *Log I/O Requests: The Log Tail.* The database log (Section 6.4) is an array of entries stored on one or more disks. As log entries are generated during transaction

processing, they are staged to an in-memory queue that is periodically flushed to the log disk(s) in FIFO order. This queue is usually called the *log tail*. In many systems, a separate process or thread is responsible for periodically flushing the log tail to the disk.

With *thread per DBMS worker*, the log tail is simply a heap-resident data structure. In the other two models, two different design choices are common. In one approach, a separate process manages the log. Log records are communicated to the log manager by shared memory or any other efficient inter-process communications protocol. In the other approach, the log tail is allocated in shared memory in much the same way as the buffer pool was handled above. The key point is that all threads and/or processes executing database client requests need to be able to request that log records be written and that the log tail be flushed.

An important type of log flush is the commit transaction flush. A transaction cannot be reported as successfully committed until a commit log record is flushed to the log device. This means that client code waits until the commit log record is flushed, and that DBMS server code must hold all resources (e.g., locks) until that time as well. Log flush requests may be postponed for a time to allow the batching of commit records in a single I/O request ("group commit").

Client communication buffers: SQL is typically used in a "pull" model: clients consume result tuples from a query cursor by repeatedly issuing the SQL `FETCH` request, which retrieve one or more tuples per request. Most DBMSs try to work ahead of the stream of FETCH requests to enqueue results in advance of client requests.

In order to support this prefetching behavior, the DBMS worker may use the client communications socket as a queue for the tuples it produces. More complex approaches implement client-side cursor caching and use the DBMS client to store results likely to be fetched

in the near future rather than relying on the OS communications buffers.

Lock table: The lock table is shared by all DBMS workers and is used by the Lock Manager (Section 6.3) to implement database locking semantics. The techniques for sharing the lock table are the same as those of the buffer pool and these same techniques can be used to support any other shared data structures needed by the DBMS implementation.

2.2 DBMS Threads

The previous section provided a simplified description of DBMS process models. We assumed the availability of high-performance OS threads and that the DBMS would target only uniprocessor systems. In the remainder of this section, we relax the first of those assumptions and describe the impact on DBMS implementations. Multi-processing and parallelism are discussed in the next section.

2.2.1 DBMS Threads

Most of today's DBMSs have their roots in research systems from the 1970s and commercialization efforts from the 1980s. Standard OS features that we take for granted today were often unavailable to DBMS developers when the original database systems were built. Efficient, high-scale OS thread support is perhaps the most significant of these. It was not until the 1990s that OS threads were widely implemented and, where they did exist, the implementations varied greatly. Even today, some OS thread implementations do not scale well enough to support all DBMS workloads well [31, 48, 93, 94].

Hence for legacy, portability, and scalability reasons, many widely used DBMS do not depend upon OS threads in their implementations. Some avoid threads altogether and use the *process per DBMS worker* or the *process pool* model. Those implementing the remaining process model choice, the *thread per DBMS worker* model, need a solution for those OS without good kernel thread implementations. One means of addressing this problem adopted by several leading DBMSs

was to implement their own proprietary, lightweight thread package. These lightweight threads, or DBMS threads, replace the role of the OS threads described in the previous section. Each DBMS thread is programmed to manage its own state, to perform all potentially blocking operations (e.g., I/Os) via non-blocking, asynchronous interfaces, and to frequently yield control to a scheduling routine that dispatches among these tasks.

Lightweight threads are an old idea that is discussed in a retrospective sense in [49], and are widely used in event-loop programming for user interfaces. The concept has been revisited frequently in the recent OS literature [31, 48, 93, 94]. This architecture provides fast task-switching and ease of porting, at the expense of replicating a good deal of OS logic in the DBMS (task-switching, thread state management, scheduling, etc.) [86].

2.3 Standard Practice

In leading DBMSs today, we find representatives of all three of the architectures we introduced in Section 2.1 and some interesting variations thereof. In this dimension, IBM DB2 is perhaps the most interesting example in that it supports four distinct process models. On OSs with good thread support, DB2 defaults to *thread per DBMS worker* and optionally supports *DBMS workers multiplexed over a thread pool*. When running on OSs without scalable thread support, DB2 defaults to *process per DBMS worker* and optionally supports *DBMS worker multiplexed over a process pool*.

Summarizing the process models supported by IBM DB2, MySQL, Oracle, PostgreSQL, and Microsoft SQL Server:

Process per DBMS worker:
This is the most straight-forward process model and is still heavily used today. DB2 defaults to *process per DBMS worker* on OSs that do not support high quality, scalable OS threads and *thread per DBMS worker* on those that do. This is also the default Oracle process model. Oracle also supports *process pool* as described below as an optional model. PostgreSQL runs the *process per DBMS worker* model exclusively on all supported operating systems.

Thread per DBMS worker: This is an efficient model with two major variants in use today:

1. *OS thread per DBMS worker*: IBM DB2 defaults to this model when running on systems with good OS thread support and this is the model used by MySQL.
2. *DBMS thread per DBMS worker*: In this model, DBMS workers are scheduled by a lightweight thread scheduler on either OS processes or OS threads. This model avoids any potential OS scheduler scaling or performance problems at the expense of high implementation costs, poor development tools support, and substantial long-standing software maintenance costs for the DBMS vendor. There are two main sub-categories of this model:
    a. *DBMS threads scheduled on OS process*: A lightweight thread scheduler is hosted by one or more OS processes. Sybase supports this model as does Informix. All current generation systems using this model implement a DBMS thread scheduler that schedules DBMS workers over multiple OS processes to exploit multiple processors. However, not all DBMSs using this model have implemented *thread migration*: the ability to reassign an existing DBMS thread to a different OS process (e.g., for load balancing).
    b. *DBMS threads scheduled on OS threads*: Microsoft SQL Server supports this model as a non-default option (default is *DBMS workers multiplexed over a thread pool* described below). This SQL Server option, called *Fibers*, is used in some high scale transaction processing benchmarks but, otherwise, is in fairly light use.

Process/thread pool:
In this model, DBMS workers are multiplexed over a pool of processes. As OS thread support has improved, a second variant of this model

has emerged based upon a *thread pool* rather than a *process pool*. In this latter model, DBMS workers are multiplexed over a pool of OS threads:

1. *DBMS workers multiplexed over a process pool*: This model is much more memory efficient than *process per DBMS worker*, is easy to port to OSs without good OS thread support, and scales very well to large numbers of users. This is the optional model supported by Oracle and the one they recommend for systems with large numbers of concurrently connected users. The Oracle default model is *process per DBMS worker*. Both of the options supported by Oracle are easy to support on the vast number of different OSs they target (at one point Oracle supported over 80 target OSs).
2. *DBMS workers multiplexed over a thread pool*: Microsoft SQL Server defaults to this model and over 99% of the SQL Server installations run this way. To efficiently support tens of thousands of concurrently connected users, as mentioned above, SQL Server optionally supports *DBMS threads scheduled on OS threads*.

As we discuss in the next section, most current generation commercial DBMSs support intra-query parallelism: the ability to execute all or parts of a single query on multiple processors in parallel. For the purposes of our discussion in this section, intra-query parallelism is the temporary assignment of multiple DBMS workers to a single SQL query. The underlying process model is not impacted by this feature in any way other than that a single client connection may have more than a single DBMS worker executing on its behalf.

2.4 Admission Control

We close this section with one remaining issue related to supporting multiple concurrent requests. As the workload in any multi-user system increases, throughput will increase up to some maximum. Beyond this point, it will begin to decrease radically as the system starts to thrash. As with OSs, thrashing is often the result of memory pressure: the

DBMS cannot keep the "working set" of database pages in the buffer pool, and spends all its time replacing pages. In DBMSs, this is particularly a problem with query processing techniques like sorting and hash joins that tend to consume large amounts of main memory. In some cases, DBMS thrashing can also occur due to contention for locks: transactions continually deadlock with each other and need to be rolled back and restarted [2]. Hence any good multi-user system has an *admission control* policy, which does not accept new work unless sufficient DBMS resources are available. With a good admission controller, a system will display *graceful degradation* under overload: transaction latencies will increase proportionally to the arrival rate, but throughput will remain at peak.

Admission control for a DBMS can be done in two tiers. First, a simple admission control policy may be in the dispatcher process to ensure that the number of client connections is kept below a threshold. This serves to prevent overconsumption of basic resources like network connections. In some DBMSs this control is not provided, under the assumption that it is handled by another tier of a multi-tier system, e.g., application servers, transaction processing monitors, or web servers.

The second layer of admission control must be implemented directly within the core DBMS relational query processor. This *execution* admission controller runs after the query is parsed and optimized, and determines whether a query is postponed, begins execution with fewer resources, or begins execution without additional constraints. The execution admission controller is aided by information from the query optimizer that estimates the resources that a query will require and the current availability of system resources. In particular, the optimizer's query plan can specify (1) the disk devices that the query will access, and an estimate of the number of random and sequential I/Os per device, (2) estimates of the CPU load of the query based on the operators in the query plan and the number of tuples to be processed, and, most importantly (3) estimates about the memory footprint of the query data structures, including space for sorting and hashing large inputs during joins and other query execution tasks. As noted above, this last metric is often the key for an admission controller, since memory pressure is typically the main cause of thrashing. Hence

many DBMSs use memory footprint and the number of active DBMS workers as the main criterion for admission control.

2.5 Discussion and Additional Material

Process model selection has a substantial influence on DBMS scaling and portability. As a consequence, three of the more broadly used commercial systems each support more than one process model across their product line. From an engineering perspective, it would clearly be much simpler to employ a single process model across all OSs and at all scaling levels. But, due to the vast diversity of usage patterns and the non-uniformity of the target OSs, each of these three DBMSs have elected to support multiple models.

Looking forward, there has been significant interest in recent years in new process models for server systems, motivated by changes in hardware bottlenecks, and by the scale and variability of workload on the Internet well [31, 48, 93, 94]. One theme emerging in these designs is to break down a server system into a set of independently scheduled "engines," with messages passed asynchronously and in bulk between these engines. This is something like the "process pool" model above, in that worker units are reused across multiple requests. The main novelty in this recent research is to break the functional granules of work in a more narrowly scoped task-specific manner than was done before. This results in many-to-many relationship between workers and SQL requests — a single query is processed via activities in *multiple* workers, and each worker does its own specialized tasks for *many* SQL requests. This architecture enables more flexible scheduling choices — e.g., it allows dynamic trade-offs between allowing a single worker to complete tasks for many queries (perhaps to improve overall system throughput), or to allow a query to make progress among multiple workers (to improve that query's latency). In some cases this has been shown to have advantages in processor cache locality, and in the ability to keep the CPU busy from idling during cache misses in hardware. Further investigation of this idea in the DBMS context is typified by the StagedDB research project [35], which is a good starting point for additional reading.

3

Parallel Architecture: Processes and Memory Coordination

Parallel hardware is a fact of life in modern servers and comes in a variety of configurations. In this section, we summarize the standard DBMS terminology (introduced in [87]), and discuss the process models and memory coordination issues in each.

3.1 Shared Memory

A *shared-memory* parallel system (Figure 3.1) is one in which all processors can access the same RAM and disk with roughly the same performance. This architecture is fairly standard today — most server hardware ships with between two and eight processors. High-end machines can ship with dozens of processors, but tend to be sold at a large premium relative to the processing resources provided. Highly parallel shared-memory machines are one of the last remaining "cash cows" in the hardware industry, and are used heavily in high-end online transaction processing applications. The cost of server hardware is usually dwarfed by costs of administering the systems, so the expense of

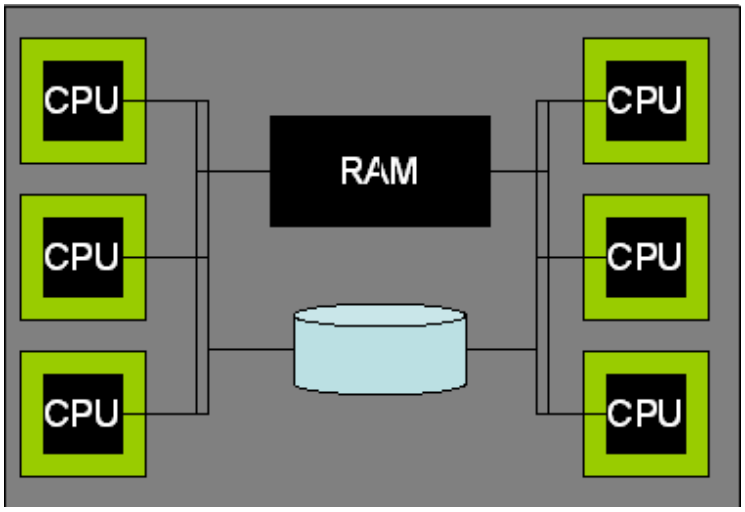


Fig. 3.1 Shared-memory architecture.

buying a smaller number of large, very expensive systems is sometimes viewed to be an acceptable trade-off.[1]

Multi-core processors support multiple processing cores on a single chip and share some infrastructure such as caches and the memory bus. This makes them quite similar to a shared-memory architecture in terms of their programming model. Today, nearly all serious database deployments involve multiple processors, with each processor having more than one CPU. DBMS architectures need to be able to fully exploit this potential parallelism. Fortunately, all three of the DBMS architectures described in Section 2 run well on modern shared-memory hardware architectures.

The process model for shared-memory machines follows quite naturally from the uniprocessor approach. In fact, most database systems evolved from their initial uniprocessor implementations to shared-memory implementations. On shared-memory machines, the OS typically supports the transparent assignment of workers (processes or

[1] The dominant cost for DBMS customers is typically paying qualified people to administer high-end systems. This includes Database Administrators (DBAs) who configure and maintain the DBMS, and System Administrators who configure and maintain the hardware and operating systems.

threads) across the processors, and the shared data structures continue to be accessible to all. All three models run well on these systems and support the execution of multiple, independent SQL requests in parallel. The main challenge is to modify the query execution layers to take advantage of the ability to parallelize a single query across multiple CPUs; we defer this to Section 5.

3.2 Shared-Nothing

A *shared-nothing* parallel system (Figure 3.2) is made up of a cluster of independent machines that communicate over a high-speed network interconnect or, increasingly frequently, over commodity networking components. There is no way for a given system to directly access the memory or disk of another system.

Shared-nothing systems provide no hardware sharing abstractions, leaving coordination of the various machines entirely in the hands of the DBMS. The most common technique employed by DBMSs to support these clusters is to run their standard process model on each machine, or node, in the cluster. Each node is capable of accepting client SQL

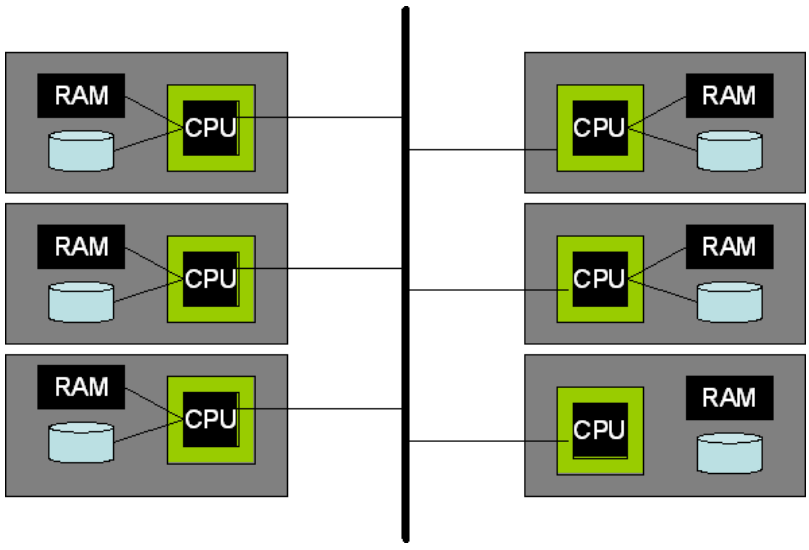


Fig. 3.2 Shared-nothing architecture.

requests, accessing necessary metadata, compiling SQL requests, and performing data access just as on a single shared memory system as described above. The main difference is that each system in the cluster stores only a portion of the data. Rather than running the queries they receive against their local data only, the requests are sent to other members of the cluster and all machines involved execute the query in parallel against the data they are storing. The tables are spread over multiple systems in the cluster using *horizontal data partitioning* to allow each processor to execute independently of the others.

Each tuple in the database is assigned to an individual machine, and hence each table is sliced "horizontally" and spread across the machines. Typical data partitioning schemes include hash-based partitioning by tuple attribute, range-based partitioning by tuple attribute, round-robin, and hybrid which is a combination of both range-based and hash-based. Each individual machine is responsible for the access, locking and logging of the data on its local disks. During query execution, the query optimizer chooses how to horizontally re-partition tables and intermediate results across the machines to satisfy the query, and it assigns each machine a logical partition of the work. The query executors on the various machines ship data requests and tuples to each other, but do not need to transfer any thread state or other low-level information. As a result of this value-based partitioning of the database tuples, minimal coordination is required in these systems. Good partitioning of the data is required, however, for good performance. This places a significant burden on the Database Administrator (DBA) to lay out tables intelligently, and on the query optimizer to do a good job partitioning the workload.

This simple partitioning solution does not handle all issues in the DBMS. For example, explicit cross-processor coordination must take place to handle transaction completion, provide load balancing, and support certain maintenance tasks. For example, the processors must exchange explicit control messages for issues like distributed deadlock detection and two-phase commit [30]. This requires additional logic, and can be a performance bottleneck if not done carefully.

Also, *partial failure* is a possibility that has to be managed in a shared-nothing system. In a shared-memory system, the failure of a

processor typically results in shutdown of the entire machine, and hence the entire DBMS. In a shared-nothing system, the failure of a single node will not necessarily affect other nodes in the cluster. But it will certainly affect the overall behavior of the DBMS, since the failed node hosts some fraction of the data in the database. There are at least three possible approaches in this scenario. The first is to bring down all nodes if any node fails; this in essence emulates what would happen in a shared-memory system. The second approach, which Informix dubbed "Data Skip," allows queries to be executed on any nodes that are up, "skipping" the data on the failed node. This is useful in scenarios where data *availability* is more important than completeness of results. But best-effort results do not have well-defined semantics, and for many workloads this is not a useful choice — particularly because the DBMS is often used as the "repository of record" in a multi-tier system, and availability-vs-consistency trade-offs tend to get done in a higher tier (often in an application server). The third approach is to employ redundancy schemes ranging from full database failover (requiring double the number of machines and software licenses) to fine-grain redundancy like *chained declustering* [43]. In this latter technique, tuple copies are spread across multiple nodes in the cluster. The advantage of chained declustering over simpler schemes is that (a) it requires fewer machines to be deployed to guarantee availability than naïve schemes, and (b) when a node does fails, the system load is distributed fairly evenly over the remaining nodes: the $n - 1$ remaining nodes each do $n/(n - 1)$ of the original work, and this form of linear degradation in performance continues as nodes fail. In practice, most current generation commercial systems are somewhere in the middle, neither as coarse-grained as full database redundancy nor as fine-grained as chained declustering

The shared-nothing architecture is fairly common today, and has unbeatable scalability and cost characteristics. It is mostly used at the extreme high end, typically for decision-support applications and data warehouses. In an interesting combination of hardware architectures, a shared-nothing cluster is often made up of many nodes each of which is a shared-memory multi-processors.

3.3 Shared-Disk

A *shared-disk* parallel system (Figure 3.3) is one in which all processors can access the disks with about the same performance, but are unable to access each other's RAM. This architecture is quite common with two prominent examples being Oracle RAC and DB2 for zSeries SYSPLEX. Shared-disk has become more common in recent years with the increasing popularity of Storage Area Networks (SAN). A SAN allows one or more logical disks to be mounted by one or more host systems making it easy to create shared disk configurations.

One potential advantage of shared-disk over shared-nothing systems is their lower cost of administration. DBAs of shared-disk systems do not have to consider partitioning tables across machines in order to achieve parallelism. But very large databases still typically do require partitioning so, at this scale, the difference becomes less pronounced. Another compelling feature of the shared-disk architecture is that the failure of a single DBMS processing node does not affect the other nodes' ability to access the entire database. This is in contrast to both shared-memory systems that fail as a unit, and shared-nothing systems that lose access to at least some data upon a node failure (unless some alternative data redundancy scheme is used). However, even with these advantages, shared-disk systems are still vulnerable to some single

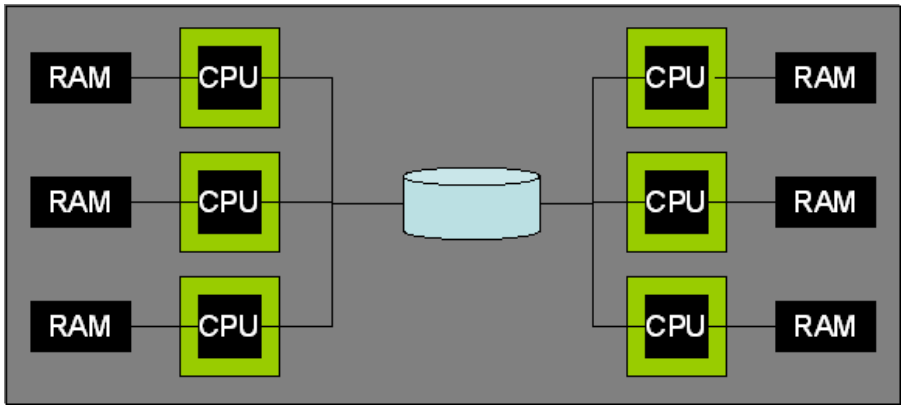


Fig. 3.3 Shared-disk architecture.

points of failure. If the data is damaged or otherwise corrupted by hardware or software failure before reaching the storage subsystem, then all nodes in the system will have access to only this corrupt page. If the storage subsystem is using RAID or other data redundancy techniques, the corrupt page will be redundantly stored but still corrupt in all copies.

Because no partitioning of the data is required in a shared-disk system, data can be copied into RAM and modified on multiple machines. Unlike shared-memory systems, there is no natural memory location to coordinate this sharing of the data — each machine has its own local memory for locks and buffer pool pages. Hence explicit coordination of data sharing across the machines is needed. Shared-disk systems depend upon a distributed lock manager facility, and a *cache-coherency* protocol for managing the distributed buffer pools [8]. These are complex software components, and can be bottlenecks for workloads with significant contention. Some systems such as the IBM zSeries SYSPLEX implement the lock manager in a hardware subsystem.

3.4 NUMA

Non-Uniform Memory Access (NUMA) systems provide a shared-memory programming model over a cluster of systems with independent memories. Each system in the cluster can access its own local memory quickly, whereas remote memory access across the high-speed cluster interconnect is somewhat delayed. The architecture name comes from this non-uniformity of memory access times.

NUMA hardware architectures are an interesting middle ground between shared-nothing and shared-memory systems. They are much easier to program than shared-nothing clusters, and also scale to more processors than shared-memory systems by avoiding shared points of contention such as shared-memory buses.

NUMA clusters have not been broadly successful commercially but one area where NUMA design concepts have been adopted is shared memory multi-processors (Section 3.1). As shared memory multi-processors have scaled up to larger numbers of processors, they have shown increasing non-uniformity in their memory architectures.

Often the memory of large shared memory multi-processors is divided into sections and each section is associated with a small subset of the processors in the system. Each combined subset of memory and CPUs is often referred to as a pod. Each processor can access local pod memory slightly faster than remote pod memory. This use of the NUMA design pattern has allowed shared memory systems to scale to very large numbers of processors. As a consequence, NUMA shared memory multi-processors are now very common whereas NUMA clusters have never achieved any significant market share.

One way that DBMSs can run on NUMA shared memory systems is by ignoring the non-uniformity of memory access. This works acceptably provided the non-uniformity is minor. When the ratio of near-memory to far-memory access times rises above the 1.5:1 to 2:1 range, the DBMS needs to employ optimizations to avoid serious memory access bottlenecks. These optimizations come in a variety of forms, but all follow the same basic approach: (a) when allocating memory for use by a processor, use memory local to that processor (avoid use of far memory) and (b) ensure that a given DBMS worker is always scheduled if possible on the same hardware processor it was on previously. This combination allows DBMS workloads to run well on high scale, shared memory systems having some non-uniformity of memory access times.

Although NUMA clusters have all but disappeared, the programming model and optimization techniques remain important to current generation DBMS systems since many high-scale shared memory systems have significant non-uniformity in their memory access performance.

3.5 DBMS Threads and Multi-processors

One potential problem that arises from implementing *thread per DBMS worker* using DBMS threads becomes immediately apparent when we remove the last of our two simplifying assumptions from Section 2.1, that of uniprocessor hardware. The natural implementation of the lightweight DBMS thread package described in Section 2.2.1 is one where all threads run within a single OS process. Unfortunately, a

single process can only be executed on one processor at a time. So, on a multi-processor system, the DBMS would only be using a single processor at a time and would leave the rest of the system idle. The early Sybase SQL Server architecture suffered this limitation. As shared memory multi-processors became more popular in the early 90s, Sybase quickly made architectural changes to exploit multiple OS processes.

When running DBMS threads within multiple processes, there will be times when one process has the bulk of the work and other processes (and therefore processors) are idle. To make this model work well under these circumstances, DBMSs must implement thread migration between processes. Informix did an excellent job of this starting with the Version 6.0 release.

When mapping DBMS threads to multiple OS processes, decisions need to be made about how many OS processes to employ, how to allocate the DBMS threads to OS threads, and how to distribute across multiple OS processes. A good rule of thumb is to have one process per physical processor. This maximizes the physical parallelism inherent in the hardware while minimizing the per-process memory overhead.

3.6 Standard Practice

With respect to support for parallelism, the trend is similar to that of the last section: most of the major DBMSs support multiple models of parallelism. Due to the commercial popularity of shared-memory systems (SMPs, multi-core systems and combinations of both), shared-memory parallelism is well-supported by all major DBMS vendors. Where we start to see divergence in support is in multi-node cluster parallelism where the broad design choices are shared-disk and shared-nothing.

- *Shared-Memory*: All major commercial DBMS providers support shared memory parallelism including: IBM DB2, Oracle, and Microsoft SQL Server.
- *Shared-Nothing*: This model is supported by IBM DB2, Informix, Tandem, and NCR Teradata among others; Green-

plum offers a custom version of PostgreSQL that supports shared-nothing parallelism.

- *Shared-Disk*: This model is supported by Oracle RAC, RDB (acquired by Oracle from Digital Equipment Corp.), and IBM DB2 for zSeries amongst others.

IBM sells multiple different DBMS products, and chose to implement shared disk support in some and shared nothing in others. Thus far, none of the leading commercial systems have support for both shared-nothing and shared-disk in a single code base; Microsoft SQL Server has implemented neither.

3.7 Discussion and Additional Material

The designs above represent a selection of hardware/software architecture models used in a variety of server systems. While they were largely pioneered in DBMSs, these ideas are gaining increasing currency in other data-intensive domains, including lower-level programmable data-processing backends like Map-Reduce [12] that are increasing users for a variety of custom data analysis tasks. However, even as these ideas are influencing computing more broadly, new questions are arising in the design of parallelism for database systems.

One key challenge for parallel software architectures in the next decade arises from the desire to exploit the new generation of "many-core" architectures that are coming from the processor vendors. These devices will introduce a new hardware design point, with dozens, hundreds or even thousands of processing units on a single chip, communicating via high-speed on-chip networks, but retaining many of the existing bottlenecks with respect to accessing off-chip memory and disk. This will result in new imbalances and bottlenecks in the memory path between disk and processors, which will almost certainly require DBMS architectures to be re-examined to meet the performance potential of the hardware.

A somewhat related architectural shift is being foreseen on a more "macro" scale, in the realm of services-oriented computing. Here, the idea is that large datacenters with tens of thousands of computers will host processing (hardware and software) for users. At this scale, appli-

cation and server administration is only affordable if highly automated. No administrative task can scale with the number of servers. And, since less reliable commodity servers are typically used and failures are more common, recovery from common failures needs to be fully automated. In services at scale there will be disk failures every day and several server failures each week. In this environment, administrative database backup is typically replaced by redundant online copies of the entire database maintained on different servers stored on different disks. Depending upon the value of the data, the redundant copy or copies may even be stored in a different datacenter. Automated offline backup may still be employed to recover from application, administrative, or user error. However, recovery from most common errors and failures is a rapid failover to a redundant online copy. Redundancy can be achieved in a number of ways: (a) replication at the data storage level (Storage-Area Networks), (b) data replication at the database storage engine level (as discussed in Section 7.4), (c) redundant execution of queries by the query processor (Section 6), or (d) redundant database requests auto-generated at the client software level (e.g., by web servers or application servers).

At a yet more decoupled level, it is quite common in practice for multiple servers with DBMS functionality to be deployed in tiers, in an effort to minimize the I/O request rate to the "DBMS of record." These schemes include various forms of middle-tier database caches for SQL queries, including specialized main-memory databases like Oracle TimesTen, and more traditional databases configured to serve this purpose (e.g., [55]). Higher up in the deployment stack, many object-oriented application-server architectures, supporting programming models like Enterprise Java Beans, can be configured to do transactional caching of application objects in concert with a DBMS. However, the selection, setup and management of these various schemes remains non-standard and complex, and elegant universally agreed-upon models have remained elusive.

4

Relational Query Processor

The previous sections stressed the macro-architectural design issues in a DBMS. We now begin a sequence of sections discussing design at a somewhat finer grain, addressing each of the main DBMS components in turn. Following our discussion in Section 1.1, we start at the top of the system with the Query Processor, and in subsequent sections move down into storage management, transactions, and utilities.

A relational query processor takes a declarative SQL statement, validates it, optimizes it into a procedural dataflow execution plan, and (subject to admission control) executes that dataflow program on behalf of a client program. The client program then fetches ("pulls") the result tuples, typically one at a time or in small batches. The major components of a relational query processor are shown in Figure 1.1. In this section, we concern ourselves with both the query processor and some non-transactional aspects of the storage manager's access methods. In general, relational query processing can be viewed as a single-user, single-threaded task. Concurrency control is managed transparently by lower layers of the system, as described in Section 5. The only exception to this rule is when the DBMS must explicitly "pin" and "unpin" buffer pool pages while operating on them so that

they remain resident in memory during brief, critical operations as we discuss in Section 4.4.5.

In this section we focus on the common-case SQL commands: Data Manipulation Language (DML) statements including SELECT, INSERT, UPDATE, and DELETE. Data Definition Language (DDL) statements such as CREATE TABLE and CREATE INDEX are typically not processed by the query optimizer. These statements are usually implemented procedurally in static DBMS logic through explicit calls to the storage engine and catalog manager (described in Section 6.1). Some products have begun optimizing a small subset of DDL statements as well and we expect this trend to continue.

4.1 Query Parsing and Authorization

Given an SQL statement, the main tasks for the SQL Parser are to (1) check that the query is correctly specified, (2) resolve names and references, (3) convert the query into the internal format used by the optimizer, and (4) verify that the user is authorized to execute the query. Some DBMSs defer some or all security checking to execution time but, even in these systems, the parser is still responsible for gathering the data needed for the execution-time security check.

Given an SQL query, the parser first considers each of the table references in the FROM clause. It canonicalizes table names into a fully qualified name of the form server.database.schema.table. This is also called a *four part* name. Systems that do not support queries spanning multiple servers need only canonicalize to database.schema.table, and systems that support only one database per DBMS can canonicalize to just schema.table. This canonicalization is required since users have context-dependent defaults that allow single part names to be used in the query specification. Some systems support multiple names for a table, called *table aliases*, and these must be substituted with the fully qualified table name as well.

After canonicalizing the table names, the query processor then invokes the *catalog manager* to check that the table is registered in the system catalog. It may also cache metadata about the table in internal query data structures during this step. Based on information about

the table, it then uses the catalog to ensure that attribute references are correct. The data types of attributes are used to drive the disambiguation logic for overloaded functional expressions, comparison operators, and constant expressions. For example, consider the expression `(EMP.salary * 1.15) < 75000`. The code for the multiplication function and comparison operator, and the assumed data type and internal format of the strings "1.15" and "75000," will depend upon the data type of the EMP.salary attribute. This data type may be an integer, a floating-point number, or a "money" value. Additional standard SQL syntax checks are also applied, including the consistent usage of tuple variables, the compatibility of tables combined via set operators (UNION/INTERSECT/EXCEPT), the usage of attributes in the SELECT list of aggregation queries, the nesting of subqueries, and so on.

If the query parses successfully, the next phase is authorization checking to ensure that the user has appropriate permissions (SELECT/DELETE/INSERT/UPDATE) on the tables, user defined functions, or other objects referenced in the query. Some systems perform full authorization checking during the statement parse phase. This, however, is not always possible. Systems that support row-level security, for example, cannot do full security checking until execution time because the security checks can be data-value dependent. Even when authorization could theoretically be statically validated at compilation time, deferring some of this work to query plan execution time has advantages. Query plans that defer security checking to execution time can be shared between users and do not require recompilation when security changes. As a consequence, some portion of security validation is typically deferred to query plan execution.

It is possible to constraint-check constant expressions during compilation as well. For example, an UPDATE command may have a clause of the form `SET EMP.salary = -1`. If an integrity constraint specifies positive values for salaries, the query need not even be executed. Deferring this work to execution time, however, is quite common.

If a query parses and passes validation, then the internal format of the query is passed on to the query rewrite module for further processing.

4.2 Query Rewrite

The query rewrite module, or rewriter, is responsible for simplifying and normalizing the query without changing its semantics. It can rely only on the query and on metadata in the catalog, and cannot access data in the tables. Although we speak of "rewriting" the query, most rewriters actually operate on an internal representation of the query, rather than on the original SQL statement text. The query rewrite module usually outputs an internal representation of the query in the same internal format that it accepted at its input.

The rewriter in many commercial systems is a logical component whose actual implementation is in either the later phases of query parsing or the early phases of query optimization. In DB2, for example, the rewriter is a stand-alone component, whereas in SQL Server the query rewriting is done as an early phase of the Query Optimizer. Nonetheless, it is useful to consider the rewriter separately, even if the explicit architectural boundary does not exist in all systems.

The rewriter's main responsibilities are:

- *View expansion*: Handling views is the rewriter's main traditional role. For each view reference that appears in the FROM clause, the rewriter retrieves the view definition from the catalog manager. It then rewrites the query to (1) replace that view with the tables and predicates referenced by the view and (2) substitute any references to that view with column references to tables in the view. This process is applied recursively until the query is expressed exclusively over tables and includes no views. This view expansion technique, first proposed for the set-based QUEL language in INGRES [85], requires some care in SQL to correctly handle duplicate elimination, nested queries, NULLs, and other tricky details [68].
- *Constant arithmetic evaluation*: Query rewrite can simplify constant arithmetic expressions: e.g., `R.x < 10+2+R.y` is rewritten as `R.x < 12+R.y`.
- *Logical rewriting of predicates*: Logical rewrites are applied based on the predicates and constants in the WHERE clause.

Simple Boolean logic is often applied to improve the match between expressions and the capabilities of index-based access methods. A predicate such as `NOT Emp.Salary > 1000000`, for example, may be rewritten as `Emp.Salary <= 1000000`. These logical rewrites can even short-circuit query execution, via simple satisfiability tests. The expression `Emp.salary < 75000 AND Emp.salary > 1000000`, for example, can be replaced with FALSE. This might allow the system to return an empty query result without accessing the database. Unsatisfiable queries may seem implausible, but recall that predicates may be "hidden" inside view definitions and unknown to the writer of the outer query. The query above, for example, may have resulted from a query for underpaid employees over a view called "Executives." Unsatisfiable predicates also form the basis for "partition elimination" in parallel installations of Microsoft SQL Server: when a relation is horizontally range partitioned across disk volumes via range predicates, the query need not be run on a volume if its range-partition predicate is unsatisfiable in conjunction with the query predicates.

An additional, important logical rewrite uses the transitivity of predicates to induce new predicates `R.x < 10 AND R.x = S.y`, for example, suggests adding the additional predicate "`AND S.y < 10`." Adding these transitive predicates increases the ability of the optimizer to choose plans that filter data early in execution, especially through the use of index-based access methods.

- *Semantic optimization*: In many cases, integrity constraints on the schema are stored in the catalog, and can be used to help rewrite some queries. An important example of such optimization is *redundant join elimination*. This arises when a foreign key constraint binds a column of one table (e.g., Emp.deptno) to another table (Dept). Given such a foreign key constraint, it is known that there is exactly one Dept for each Emp and that the Emp tuple could not exist without a corresponding Dept tuple (the parent).

Consider a query that joins the two tables but does not make use of the Dept columns:

```
SELECT Emp.name, Emp.salary
  FROM Emp, Dept
  WHERE Emp.deptno = Dept.dno
```

Such queries can be rewritten to remove the Dept table (assuming Emp.deptno is constrained to be non-null), and hence the join. Again, such seemingly implausible scenarios often arise naturally via views. A user, for example, may submit a query about employee attributes over a view EMPDEPT that joins those two tables. Database applications such as Siebel use very wide tables and, where the underlying database does not support tables of sufficient width, they use multiple tables with a view over these tables. Without redundant join elimination, this view-based wide-table implementation would perform very poorly.

Semantic optimizations can also circumvent query execution entirely when constraints on the tables are incompatible with query predicates.

- *Subquery flattening and other heuristic rewrites*: Query optimizers are among the most complex components in current-generation commercial DBMSs. To keep that complexity bounded, most optimizers operate on individual SELECT-FROM-WHERE query blocks in isolation and do not optimize across blocks. So rather than further complicate query optimizers, many systems rewrite queries into a form better suited for the optimizer. This transformation is sometimes called query normalization. One example class of normalizations is to rewrite semantically equivalent queries into a canonical form, in an effort to ensure that semantically equivalent queries will be optimized to produce the same query plan. Another important heuristic is to flatten nested queries when possible to maximally expose opportunities for the query optimizer's single-block optimizations. This turns out to be very tricky in some cases in SQL, due to issues

like duplicate semantics, subqueries, NULLs, and correlation [68, 80]. In the early days, subquery flattening was a purely heuristic rewrite but some products are now basing the rewrite decision on cost-based analysis. Other rewrites are possible across query blocks as well. Predicate transitivity, for example, can allow predicates to be copied across subqueries [52]. Flattening correlated subqueries is especially important to achieve good performance in parallel architectures: correlated subqueries result in "nested-loop" style comparisons across query blocks, which serializes the execution of the subqueries despite the availability of parallel resources.

4.3 Query Optimizer

The query optimizer's job is to transform an internal query representation into an efficient *query plan* for executing the query (Figure 4.1). A query plan can be thought of as a dataflow diagram that pipes table data through a graph of query *operators*. In many systems, queries are first broken into SELECT-FROM-WHERE query blocks. The optimization of each individual query block is then done using techniques similar to those described in the famous paper by Selinger *et al.* on the System R optimizer [79]. On completion, a few operators are typically added to the top of each query block as post-processing to compute GROUP BY, ORDER BY, HAVING and DISTINCT clauses if they exist. The various blocks are then stitched together in a straightforward fashion.

The resulting query plan can be represented in a number of ways. The original System R prototype compiled query plans into machine code, whereas the early INGRES prototype generated an interpretable query plan. Query interpretation was listed as a "mistake" by the INGRES authors in their retrospective paper in the early 1980's [85], but Moore's law and software engineering have vindicated the INGRES decision to some degree. Ironically, compiling to machine code is listed by some researchers on the System R project as a mistake. When the System R code base was made into a commercial DBMS system

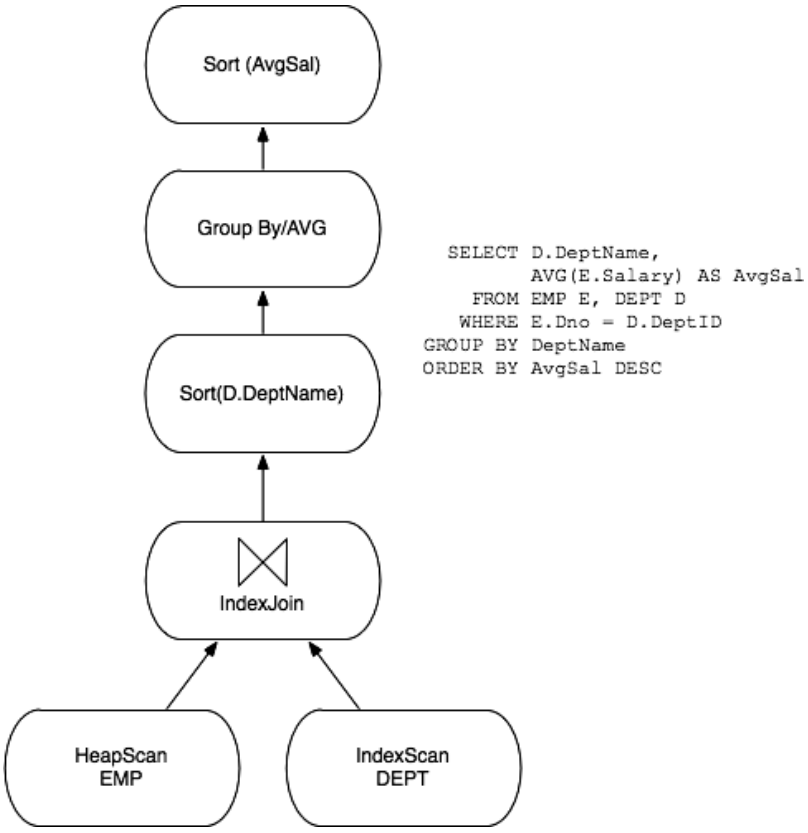


Fig. 4.1 A Query plan. Only the main physical operators are shown.

(SQL/DS) the development team's first change was to replace the machine code executor with an interpreter.

To enable cross-platform portability, every major DBMS now compiles queries into some kind of interpretable data structure. The only difference between them is the intermediate form's level of abstraction. The query plan in some systems is a very lightweight object, not unlike a relational algebraic expression, that is annotated with the names of access methods, join algorithms, and so on. Other systems use a lower-level language of "op-codes," closer in spirit to Java byte codes than to relational algebraic expressions. For simplicity in our discussion, we focus on algebra-like query representations in the remainder of this paper.

Although Selinger's paper is widely considered the "bible" of query optimization, it was preliminary research. All systems extend this work significantly in a number of dimensions. Among the main extensions are:

- *Plan space*: The System R optimizer constrained its plan space somewhat by focusing only on "left-deep" query plans (where the right-hand input to a join must be a base table), and by "postponing Cartesian products" (ensuring that Cartesian products appear only after all joins in a dataflow). In commercial systems today, it is well known that "bushy" trees (with nested right-hand inputs) and early use of Cartesian products can be useful in some cases. Hence both options are considered under some circumstances by most systems.
- *Selectivity estimation*: The selectivity estimation techniques in the Selinger paper are based on simple table and index cardinalities and are naïve by the standards of current generation systems. Most systems today analyze and summarize the distributions of values in attributes via histograms and other summary statistics. Since this involves visiting every value in each column, it can be relatively expensive. Consequently, some systems use sampling techniques to get an estimation of the distribution without the expense of an exhaustive scan.

  Selectivity estimates for joins of base tables can be made by "joining" the histograms on the join columns. To move beyond single-column histograms, more sophisticated schemes to incorporate issues like dependencies among columns have recently been proposed [16, 69]. These innovations have begun to show up in commercial products, but considerable progress remains to be made. One reason for the slow adoption of these schemes was a longstanding flaw in many industry benchmarks: the data generators in benchmarks like TPC-D and TPC-H generated statistically independent values in columns, and hence did not encourage the adoption of technology to handle "real" data distributions. This benchmark flaw has been addressed in the TPC-DS

benchmark [70]. Despite slow adoption rates, the benefits of improved selectivity estimation are widely recognized. Ioannidis and Christodoulakis noted that errors in selectivity early in optimization propagate multiplicatively up the plan tree and result in terrible subsequent estimations [45].

- *Search Algorithms*: Some commercial systems, notably those of Microsoft and Tandem, discard Selinger's dynamic programming optimization approach in favor of a goal-directed "top-down" search scheme based on the techniques used in Cascades [25]. Top-down search can in some instances lower the number of plans considered by an optimizer [82], but can also have the negative effect of increasing optimizer memory consumption. If practical success is an indication of quality, then the choice between top-down search and dynamic programming is irrelevant. Each has been shown to work well in state-of-the-art optimizers, and both still have runtimes and memory requirements that are, unfortunately, exponential in the number of tables in a query.

  Some systems fall back on heuristic search schemes for queries with "too many" tables. Although the research literature of randomized query optimization heuristics is interesting [5, 18, 44, 84], the heuristics used in commercial systems tend to be proprietary, and apparently do not resemble the randomized query optimization literature. An educational exercise is to examine the query "optimizer" of the open-source MySQL engine, which at last check was entirely heuristic and relied mostly on exploiting indexes and key/foreign-key constraints. This is reminiscent of early (and infamous) versions of Oracle. In some systems, a query with too many tables in the FROM clause can only be executed if the user explicitly directs the optimizer how to choose a plan (via so-called optimizer "hints" embedded in the SQL).
- *Parallelism*: Every major commercial DBMS today has some support for parallel processing. Most also support "intra-query" parallelism: the ability to speed up a single query via the use of multiple processors. The query optimizer needs

to get involved in determining how to schedule operators — and parallelized operators — across multiple CPUs, and (in the shared-nothing or shared-disk cases) across multiple separate computers. Hong and Stonebraker [42] chose to avoid the parallel optimization complexity problem and use two phases: first a traditional single-system optimizer is invoked to pick the best single-system plan, and then this plan is scheduled across multiple processors or machines. Research has been published on this second optimization phase [19, 21] although it is not clear to what extent these results have influenced current practice.

Some commercial systems implement the two-phase approach described above. Others attempt to model the cluster network topology and data distribution across the cluster to produce an optimal plan in a single phase. While the single pass approach can be shown to produce better plans under some circumstances, it's not clear whether the additional query plan quality possible using a single phase approach justifies the additional optimizer complexity. Consequently, many current implementations still favor the two-phase approach. Currently this area seems to be more like art than science. The Oracle OPS (now called RAC) shared-disk cluster uses a two phase optimizer. IBM DB2 Parallel Edition (now called DB2 Database Partitioning Feature) was first implemented using a two-phase optimizer but has since been evolving toward a single-phase implementation.

- *Auto-Tuning*: A variety of ongoing industrial research efforts attempt to improve the ability of a DBMS to make tuning decisions automatically. Some of these techniques are based on collecting a query workload, and then using the optimizer to find the plan costs via various "what-if" analyses. What if, for example, other indexes had existed or the data had been laid out differently? An optimizer needs to be adjusted somewhat to support this activity efficiently, as described by Chaudhuri and Narasayya [12]. The Learning Optimizer (LEO) work of Markl et al. [57] is also in this vein.

4.3.1 A Note on Query Compilation and Recompilation

SQL supports the ability to "prepare" a query: to pass it through the parser, rewriter and, optimizer, store the resulting query execution plan, and use it in subsequent "execute" statements. This is even possible for dynamic queries (e.g., from web forms) that have program variables in the place of query constants. The only wrinkle is that during selectivity estimation, the variables that are provided by the forms are assumed by the optimizer to take on "typical" values. When non-representative "typical" values are chosen, extremely poor query execution plans can result. Query preparation is especially useful for form-driven, canned queries over fairly predictable data: the query is prepared when the application is written, and when the application goes live, users do not experience the overhead of parsing, rewriting, and optimizing.

Although preparing a query when an application is written can improve performance, this is a very restrictive application model. Many application programmers, as well as toolkits like Ruby on Rails, build SQL statements dynamically during program execution, so precompiling is not an option. Because this is so common, DBMSs store these dynamic query execution plans in the query plan cache. If the same (or very similar) statement is subsequently submitted, the cached version is used. This technique approximates the performance of precompiled static SQL without the application model restrictions and is heavily used.

As a database changes over time, it often becomes necessary to re-optimize prepared plans. At a minimum, when an index is dropped, any plan that used that index must be removed from the stored plan cache, so that a new plan will be chosen upon the next invocation.

Other decisions about re-optimizing plans are more subtle, and expose philosophical distinctions among the vendors. Some vendors (e.g., IBM) work very hard to provide *predictable performance* across invocations at the expense of optimal performance per invocation. As a result, they will not re-optimize a plan unless it will no longer execute, as in the case of dropped indexes. Other vendors (e.g., Microsoft) work very hard to make their systems *self-tuning*, and will re-optimize plans

more aggressively. For example, if the cardinality of a table changes significantly, recompilation will be triggered in SQL Server since this change may influence the optimal use of indexes and join orders. A self-tuning system is arguably less predictable, but more efficient in a dynamic environment.

This philosophical distinction arises from differences in the historical customer base for these products. IBM traditionally focused on high-end customers with skilled DBAs and application programmers. In these kinds of high-budget IT shops, predictable performance from the database is of paramount importance. After spending months tuning the database design and settings, the DBA does not want the optimizer to change it unpredictably. By contrast, Microsoft strategically entered the database market at the low end. As a result, their customers tend to have lower IT budgets and expertise, and want the DBMS to "tune itself" as much as possible.

Over time these companies' business strategies and customer bases have converged so that they compete directly, and their approaches are evolving together. Microsoft has high-scale enterprise customers that want complete control and query plan stability. And IBM has some customers without DBA resources needing full auto-administration.

4.4 Query Executor

The query executor operates on a fully-specified query plan. This is typically a directed dataflow graph that connects operators that encapsulate base-table access and various query execution algorithms. In some systems, this dataflow graph is already compiled into low-level op-codes by the optimizer. In this case, the query executor is basically a runtime interpreter. In other systems, the query executor receives a representation of the dataflow graph and recursively invokes procedures for the operators based on the graph layout. We focus on this latter case, as the op-code approach essentially compiles the logic we describe here into a program.

Most modern query executors employ the *iterator* model that was used in the earliest relational systems. Iterators are most simply described in an object-oriented fashion. Figure 4.2 shows a simplified definition for an iterator . Each iterator specifies its inputs that define

```
class iterator {
  iterator &inputs[];
  void init();
  tuple get_next();
  void close();
}
```

Fig. 4.2 Iterator superclass pseudocode.

the edges in the dataflow graph. All operators in a query plan — the nodes in the dataflow graph — are implemented as subclasses of the iterator class. The set of subclasses in a typical system might include filescan, indexscan, sort, nested-loops join, merge-join, hash-join, duplicate-elimination, and grouped-aggregation. An important feature of the iterator model is that any subclass of iterator can be used as input to any other. Hence each iterator's logic is independent of its children and parents in the graph, and special-case code for particular combinations of iterators is not needed.

Graefe provides more details on iterators in his query execution survey [24]. The interested reader is also encouraged to examine the open-source PostgreSQL code base. PostgreSQL utilizes moderately sophisticated implementations of the iterators for most standard query execution algorithms.

4.4.1 Iterator Discussion

An important property of iterators is that they *couple dataflow with control flow*. The `get_next()` call is a standard procedure call that returns a tuple reference to the caller via the call stack. Hence a tuple is returned to a parent in the graph exactly when control is returned. This implies that only a single DBMS thread is needed to execute an entire query graph, and queues or rate-matching between iterators are not needed. This makes relational query executors clean to implement and easy to debug, and is a contrast with dataflow architectures in other environments. Networks, for example, rely on various protocols for queuing and feedback between concurrent producers and consumers.

The single-threaded iterator architecture is also quite efficient for single-system (non-cluster) query execution. In most database applica-

tions, the performance metric of merit is *time to query completion*, but other optimization goals are possible. For example, maximizing DBMS throughput is another reasonable goal. Another that is popular with interactive applications is *time to first row*. In a single-processor environment, time to completion for a given query plan is achieved when resources are fully utilized. In an iterator model, since one of the iterators is always active, resource utilization is maximized.[1]

As we mentioned previously, most modern DBMSs support parallel query execution. Fortunately, this support can be provided with essentially no changes to the iterator model or query execution architecture. Parallelism and network communications can be encapsulated within special *exchange* iterators, as described by Graefe [23]; these also implement network-style "pushing" of data in a manner that is invisible to the DBMS iterators, which retain a "pull"-style `get_next()` API. Some systems make the push logic explicit in their query execution model as well.

4.4.2 Where's the Data?

Our discussion of iterators has conveniently sidestepped any questions of memory allocation for in-flight data. We neither specified how tuples were stored in memory, nor how they were passed between iterators. In practice, each iterator is pre-allocated a fixed number of *tuple descriptors*, one for each of its inputs, and one for its output. A tuple descriptor is typically an array of column references, where each column reference is composed of a reference to a tuple somewhere else in memory, and a column offset in that tuple. The basic iterator superclass logic never allocates memory dynamically. This raises the question of where the actual tuples being referenced are stored in memory.

There are two possible answers to this question. The first is that tuples reside in pages in the buffer pool. We call these *BP-tuples*. If an iterator constructs a tuple descriptor that references a BP-tuple, it must

[1] This assumes that iterators never block waiting for I/O requests. In the cases where prefetching is ineffective, inefficiencies in the iterator model can occur due to blocking on I/O. This is typically not a problem in single-system databases, but it arises frequently when executing queries over remote tables or in multi-system clusters [23, 56].

increment the *pin count* of the tuple's page — a count of the number of active references to tuples on that page. It decrements the pin count when the tuple descriptor is cleared. The second possibility is that an iterator implementation may allocate space for a tuple on the memory heap. We call this an *M-tuple*. An iterator may construct an M-tuple by copying columns from the buffer pool (the copy bracketed by a pin increment/decrement pair), and/or by evaluating expressions (e.g., arithmetic expressions like "EMP.sal $*$ 0.1") in the query specification.

One general approach is to always copy data out of the buffer pool immediately into M-tuples. This design uses M-tuples as the only in-flight tuple structure and simplifies the executor code. The design also circumvents bugs that can result from having buffer-pool pin and unpin calls separated by long periods of execution (and many lines of code). One common bug of this sort is to forget to unpin the page altogether (a "buffer leak"). Unfortunately, as noted in Section 4.2, exclusive use of M-tuples can be a major performance problem, since memory copies are often a serious bottleneck in high-performance systems.

On the other hand, constructing an M-tuple makes sense in some cases. As long as a BP-tuple is directly referenced by an iterator, the page on which the BP-tuple resides must remain pinned in the buffer pool. This consumes a page worth of buffer pool memory, and ties the hands of the buffer replacement policy. Copying a tuple out of the buffer pool can be beneficial if the tuple will continue to be referenced for a long period of time.

The upshot of this discussion is that the most efficient approach is to support tuple descriptors that can reference both BP-tuples and M-tuples.

4.4.3 Data Modification Statements

Up to this point we have only discussed queries, that is, read-only SQL statements. Another class of DML statements exist that modify data: INSERT, DELETE, and UPDATE statements. Execution plans for these statements typically look like simple straight-line query plans, with a single access method as the source, and a data modification operator at the end of the pipeline.

In some cases, however, these plans both query and modify the same data. This mix of reading and writing the same table (possibly multiple times) requires some care. A simple example is the notorious "Halloween problem,"[2] so called because it was discovered on October 31st by the System R group. The Halloween problem arises from a particular execution strategy for statements like "give everyone whose salary is under $20K a 10% raise." A naïve plan for this query pipelines an index scan iterator over the Emp.salary field into an update iterator (the left-hand side of Figure 4.3). The pipelining provides good I/O

```
UPDATE EMP
   SET salary=salary*1.1
 WHERE salary < 20000
```

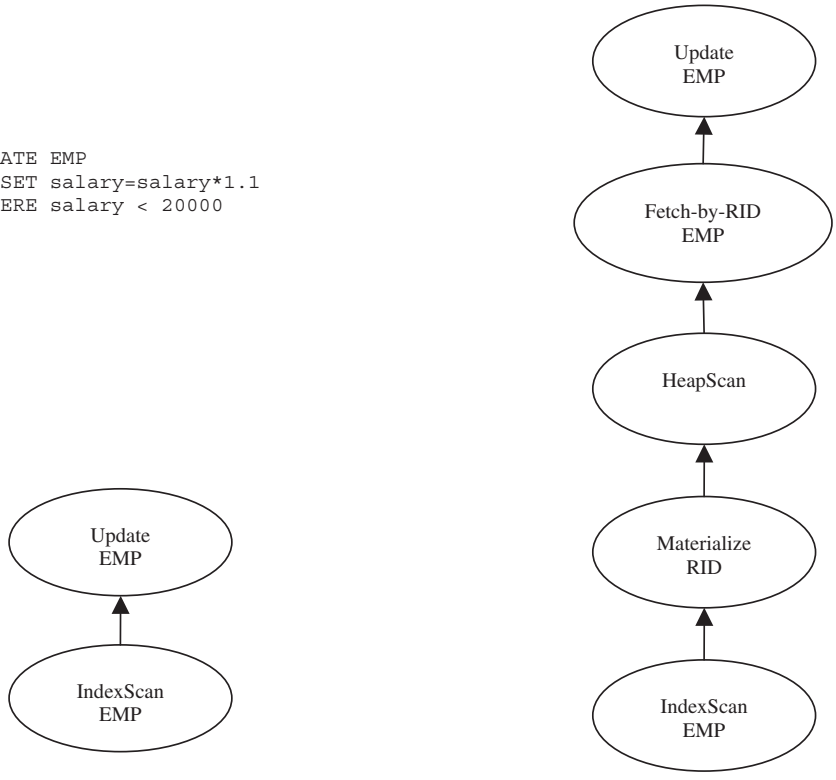


Fig. 4.3 Two query plans for updating a table via an IndexScan. The plan on the left is susceptible to the Halloween problem. The plan on the right is safe, since it identifies all tuples to be updated before actually performing any updates.

[2] Despite the spooky similarity in names, the Halloween problem has nothing to do with the phantom problem of Section 4.2.1

locality, because it modifies tuples just after they are fetched from the B+-tree. This pipelining, however, can also result in the index scan "rediscovering" a previously modified tuple that moved rightward in the tree after modification, leading to multiple raises for each employee. In our example, all low-paid employees will receive repeated raises until they earn more than $20K. This is not the intention of the statement.

SQL semantics forbid this behavior: a single SQL statement is not allowed to "see" its own updates. Some care is needed to ensure that this visibility rule is observed. A simple, safe implementation has the query optimizer choose plans that avoid indexes on the updated column. This can be quite inefficient in some cases. Another technique is to use a batch read-then-write scheme. This interposes Record-ID materialization and fetching operators between the index scan and the data modification operators in the dataflow (right-hand side of Figure 4.3). The materialization operator receives the IDs of all tuples to be modified and stores them in temporary file. It then scans the temporary file and fetches each physical tuple ID by RID and feeds the resulting tuple to the data modification operator. If the optimizer chooses an index, in most cases this implies that only a few tuples are being changed. Hence the apparent inefficiency of this technique may be acceptable, since the temporary table is likely to remain entirely in the buffer pool. Pipelined update schemes are also possible, but require (somewhat exotic) multi-version support from the storage engine [74].

4.5 Access Methods

Access methods are the routines that manage access to the various disk-based data structures that the system supports. These typically included unordered files ("heaps"), and various kinds of indexes. All major commercial systems implement heaps and B+-tree indexes. Both Oracle and PostgreSQL support hash indexes for equality lookups. Some systems are beginning to introduce rudimentary support for multi-dimensional indexes such as R-trees [32]. PostgreSQL supports an extensible index called the Generalized Search Tree (GiST) [39], and currently uses it to implement R-trees for multi-dimensional data, and RD-trees for text data [40]. IBM UDB Version 8 introduced

Multi-Dimensional Clustered (MDC) Indexes for accessing data via ranges on multiple dimensions [66]. Systems that target read-mostly data warehousing workloads often include specialized bitmap variants of indexes as well [65], as we describe in Section 4.6.

The basic API that an access method provides is an iterator API. The `init()` routine is expanded to accept a "search argument" (or in the terminology of System R, a SARG) of the form `column operator constant`. A `NULL` SARG is treated as a request to scan all tuples in the table. The `get_next()` call at the access method layer returns `NULL` when no more tuples satisfy the search argument.

There are two reasons to pass SARGs into the access method layer. The first reason should be clear: index access methods like B+-trees require SARGs in order to function efficiently. The second reason is a more subtle performance issue, but one that applies to heap scans as well as index scans. Assume that the SARG is checked by the routine that calls the access method layer. Then each time the access method returns from `get_next()`, it must either (a) return a handle to a tuple residing in a frame in the buffer pool, and pin the page in that frame to avoid replacement or (b) make a copy of the tuple. If the caller finds that the SARG is not satisfied, it is responsible for either (a) decrementing the pin count on the page, or (b) deleting the copied tuple. It must then reinvoke `get_next()`to try the next tuple on the page. This logic consumes a significant number of CPU cycles in function call/return pairs, and will either pin pages in the buffer pool unnecessarily (generating unnecessary contention for buffer frames) or create and destroy copies of tuples unnecessarily — a significant CPU overhead when streaming through millions of tuples. Note that a typical heap scan will access all of the tuples on a given page, resulting in multiple iterations of this interaction per page. By contrast, if all this logic is done in the access method layer, the repeated pairs of call/return and either pin/unpin or copy/delete can be avoided by testing the SARGs a page at a time, and only returning from a `get_next()` call for a tuple that satisfies the SARG. SARGS keep a nice clean architectural boundary between the storage engine and the relational engine while obtaining excellent performance. Consequently, many systems support very rich SARG support and use them broadly. Thematically, this is

an instance of the standard DBMS wisdom of amortizing work across multiple items in a collection, but in this case it is being applied for CPU performance, rather than disk performance.

All DBMSs need some way to "point" to rows in a base table, so that index entries can reference the rows appropriately. In many DBMSs, this is implemented by using direct row IDs (RIDs) that are the physical disk addresses of the rows in the base tables. This has the advantage of being fast, but has the downside of making base table row movement very expensive since all secondary indexes that point to this row require updating. Both finding and updating these rows can be costly. Rows need to move when an update changes the row size and space is unavailable on the current page for the freshly updated row. And many rows need to move when a B+-tree is split. DB2 uses a forwarding pointer to avoid the first problem. This requires a second I/O to find a moved page, but avoids having to update the secondary index. DB2 avoids the second problem by simply not supporting B+-trees as primary storage for base table tuples. Microsoft SQL Server and Oracle support B+-trees as primary storage and must be able to deal with row movement efficiently. The approach taken is to avoid using a physical row address in the secondary indexes and instead use the row primary key (with some additional system provided bits to force uniqueness if the table does not have a unique key) rather than the physical RID. This sacrifices some performance when using a secondary index to access a base table row, but avoids the problems that row movement causes. Oracle avoids the performance penalty of this approach in some cases by keeping a physical pointer with the primary key. If the row has not moved, it will be found quickly using the physical pointer. But, if it has moved, the slower primary key technique will be used.

Oracle avoids moving rows in heap files by allowing rows to span pages. So, when a row is updated to a longer value that no longer fits on the original page, rather than being forced to move the row, they store what fits in the original page and the remainder can span to the next.

In contrast to all other iterators, access methods have deep interactions with the concurrency and recovery logic surrounding transactions, as described in Section 4.

4.6 Data Warehouses

Data Warehouses — large historical databases for decision-support that are loaded with new data on a periodic basis — have evolved to require specialized query processing support, and in the next section we survey some of the key features that they tend to require. This topic is relevant for two main reasons:

1. Data warehouses are a very important application of DBMS technology. Some claim that warehouses account for 1/3 of all DBMS activity [26, 63].
2. The conventional query optimization and execution engines discussed so far in this section do not work well on data warehouses. Hence, extensions or modifications are required to achieve good performance.

Relational DBMSs were first architected in the 1970's and 1980's to address the needs of business data processing applications, since that was the dominant requirement at the time. In the early 1990's the market for data warehouses and "business analytics" appeared, and has grown dramatically since that time.

By the 1990's on-line transaction processing (OLTP) had replaced batch business data processing as the dominant paradigm for database usage. Moreover, most OLTP systems had banks of computer operators submitting transactions, either from phone conversations with the end customer or by performing data entry from paper. Automated teller machines had become widespread, allowing customers to do certain interactions directly without operator intervention. Response time for such transactions was crucial to productivity. Such response time requirements have only become more urgent and varied today as the web is fast replacing operators with self service by the end customer.

About the same time, enterprises in the retail space had the idea to capture all historical sales transactions, and to store them typically for one or two years. Such historical sales data can be used by buyers to figure out "what's hot and what's not." Such information can be leveraged to affect purchasing patterns. Similarly, such data can be used to decide what items to put on promotion, which ones to discount, and

which ones to send back to the manufacturer. The common wisdom of the time was that a historical data warehouse in the retail space paid for itself through better stock management, shelf and store layout in a matter of months.

It was clear at the time that a data warehouse should be deployed on separate hardware from an OLTP system. Using that methodology, the lengthy (and often unpredictable) business intelligence queries would not spoil OLTP response time. Also, the nature of data is very different; warehouses deal with history, OLTP deals with "now." Finally, it was found that the schema desired for historical data often did not match the schema desired for current data, and a data transformation was required to convert from one to the other.

For these reasons workflow systems were constructed that would "scrape" data from operational OLTP systems and load it into a data warehouse. Such systems were branded "extract, transform, and load" (ETL) systems. Popular ETL products include Data Stage from IBM and PowerCenter from Informatica. In the last decade ETL vendors have extended their products with data cleansing tools, de-duplication tools, and other quality-oriented offerings.

There are several issues that must be dealt with in the data warehouse environment, as we discuss below.

4.6.1 Bitmap Indexes

B+-trees are optimized for fast insertion, deletion, and update of records. In contrast, a data warehouse performs an initial load and then the data is static for months or years. Moreover, data warehouses often have columns with a small number of values. Consider, for example, storing the sex of a customer. There are only two values, and this can be represented by one bit per record in a bitmap. In contrast, a B+-tree will require (value, record-pointer) pairs for each record and will typically consume 40 bits per record.

Bitmaps are also advantageous for conjunctive filters, such as

Customer.sex = "F" and Customer.state = "California"

In this case, the result set can be determined by intersecting bitmaps. There are a host of more sophisticated bitmap arithmetic

tricks that can be played to improve performance of common analytics queries. For a discussion of bitmap processing, the interested reader should consult [65].

In current products, bitmap indexes complement the B+-trees in Oracle for indexing stored data, while DB2 offers a more limited version. Sybase IQ makes extensive use of bitmap indexes. Of course, the disadvantage of bitmaps is that they are expensive to update, so their utility is restricted to warehouse environments.

4.6.2 Fast Load

Often, data warehouses are loaded in the middle of the night with the days' transactional data. This was an obvious strategy for retail establishments that are open only during the day. A second reason for bulk nightly loads is to avoid having updates appear during user interaction. Consider the case of a business analyst who wishes to formulate some sort of *ad-hoc* query, perhaps to investigate the impact of hurricanes on customer buying patterns. The result of this query might suggest a follow-on query, such as investigating buying patterns during large storms. The result of these two queries should be compatible, i.e., the answers should be computed on the same data set. This can be problematic for queries that include recent history, if data is being concurrently loaded.

As such, it is crucial that data warehouses be bulk-loadable very quickly. Although one could program warehouse loads with a sequence of SQL insert statements, this tactic is never used in practice. Instead a bulk loader is utilized that will stream large numbers of records into storage without the overhead of the SQL layer, and taking advantage of special bulk-load methods for access methods like B+-trees. In round numbers, a bulk loader is an order of magnitude faster than SQL inserts, and all major vendors offer a high performance bulk loader.

As the world moves to e-commerce and 24 hour-per-day sales, this bulk load tactic makes less sense. But the move to "real time" warehouses has a couple of problems. First, inserts, whether from the bulk loader or from transactions, must set write locks as discussed in Section 6.3. These collide with the read locks acquired by

queries, and can cause the warehouse to "freeze up." Second, providing compatible answers across query sets as described above is problematic.

Both issues can be circumvented by avoiding update-in-place, and providing historical queries. If one keeps before and after values of updates, suitably timestamped, then one can provide queries as of a time in the recent past. Running a collection of queries as of the same historical time will provide compatible answers. Moreover, the same historical queries can be run without setting read locks.

As discussed in Section 5.2.1, multi-version (MVCC) isolation levels like SNAPSHOT ISOLATION are provided by some vendors, notably Oracle. As real time warehouses become more popular, the other vendors will presumably follow suit.

4.6.3 Materialized Views

Data warehouses are typically gigantic, and queries that join multiple large tables have a tendency to run "forever." To speed up performance on popular queries, most vendors offer materialized views. Unlike the purely logical views discussed earlier in this section, materialized views take are actual tables that can be queried, but which correspond to a logical view expression over the true "base" data tables. A query to a materialized view will avoid having to perform the joins in the view expression at run time. Instead, the materialized view must be kept up to date as updates are performed.

There are three aspects to Materialized View use: (a) selecting the views to materialize, (b) maintaining the freshness of the views, and (c) considering the use of materialized views in *ad-hoc* queries. Topic (a) is an advanced aspect of the automatic database tuning we mentioned in Section 4.3. Topic (c) is implemented to varying extents in the various products; the problem is theoretically challenging even for simple single-block queries [51], and moreso for generic SQL with aggregation and subqueries. For (b), most vendors offer multiple refresh techniques, ranging from performing a materialized view update on each update to the tables from which the materialized view is derived, to periodically discarding and then recreating the materialized view.

Such tactics offer a trade-off between run-time overhead and data consistency of the materialized view.

4.6.4 OLAP and *Ad-hoc* Query Support

Some warehouse workloads have predictable queries. For example, at the end of every month, a summary report might be run to provide total sales by department for each sales region in a retail chain. Interspersed with this workload are *ad-hoc queries*, formulated on the fly by business analysts.

Obviously, predictable queries can be supported by appropriately constructed materialized views. More generally, since most business analytics queries ask for aggregates, one can compute a materialized view which is the aggregate sales by department for each store. Then, if the above region query is specified, it can be satisfied by "rolling up" individual stores in each region.

Such aggregates are often called *data cubes*, and are an interesting class of materialized views. In the early 1990's products such as Essbase provided customized tools for storing data in priority cube format while providing cube-based user interfaces to navigate the data, and this capability came to be known as on-line analytical processing (OLAP). Over time, data cube support has been added to full-function relational database systems, and is often termed Relational OLAP (ROLAP). Many DBMSs that provide ROLAP have evolved to internally implement some of the earlier OLAP-style storage schemes in special cases, and as a result are sometimes referred to as HOLAP (Hybrid OLAP) schemes.

Clearly, data cubes provide high performance for a predictable, limited class of queries. However, they are generally not helpful for supporting *ad-hoc* queries.

4.6.5 Optimization of Snowflake Schema Queries

Many data warehouses follow a particular method for schema design. Specifically, they store a collection of *facts*, which, in a retail environment, are typically simple records like "customer X bought product Y from store Z at time T." A central fact table records infor-

mation about each fact, such as the purchase price, discount, sales tax information, etc. Also in the fact table are foreign keys for each of a set of *dimensions*. Dimensions can include customers, products, stores, time, etc. A schema of this form is usually called a *star schema,* because it has a central fact table surrounded by dimensions, each with a 1-N primary-key-foreign-key relationship to the fact table. Drawn in an entity-relationship diagram, such a schema is star-shaped.

Many dimensions are naturally hierarchical. For example, if stores can be aggregated into regions, then the Stores "dimension table" has an added foreign key to a Region dimension table. Similar hierarchies are typical for attributes involving time (months/days/years), management hierarchies, and so on. In these cases, a multi-level star, or *snowflake* schema results.

Essentially all data warehouse queries entail filtering one or more dimensions in a snowflake schema on some attributes in these tables, then joining the result to the central fact table, grouping by some attributes in the fact table or a dimension table, and then computing an SQL aggregate.

Over time, vendors have special-cased this class of query in their optimizers, because it is so popular, and it is crucial to choose a good plan for such long running commands.

4.6.6 Data Warehousing: Conclusions

As can be seen, data warehouses require quite different capabilities from OLTP environments. In addition to B+-trees, one needs bitmap indexes. Instead of a general purpose optimizer, one needs to focus special attention on aggregate queries over snowflake schemas. Instead of normal views, one requires materialized views. Instead of fast transactional updates, one needs fast bulk load, etc. A longer overview of data warehousing practices can be found in [11].

The major relational vendors began with OLTP-oriented architectures, and have added warehouse-oriented features over time. In addition, there are a variety of smaller vendors offering DBMS solutions in this space. These include Teradata and Netezza, who offer shared-nothing proprietary hardware on which their DBMSs run. Also, selling

to this space are Greenplum (a parallelization of PostgreSQL), DATAllegro, and EnterpriseDB, all of whom are run on more conventional hardware.

Finally, there are some (including one of the authors) who claim that *column stores* have a huge advantage in the data warehouse space versus traditional storage engines in which the unit of storage is a table row. Storing each column separately is especially efficient when tables are "wide" (high arity), and accesses tend to be on only a few columns. Column storage also enables simple and effective disk compression, since all data in a column is from the same type. The challenge with column stores is that the position of rows in the table needs to remain consistent across all stored columns, or extra mechanisms are needed to join up columns. This is a big problem for OLTP, but not a major issue for append-mostly databases like warehouses or system-log repositories. Vendors offering column stores include Sybase, Vertica, Sand, Vhayu, and KX. More details on this architecture discussion can be found in [36, 89, 90].

4.7 Database Extensibility

Relational databases have traditionally been viewed as being limited in the kinds of data they store, focused mostly on the "facts and figures" used in corporate and administrative record-keeping. Today, however, they host a wide range of data types expressed in a variety of popular programming languages. This is achieved by making the core relational DBMS extensible in a variety of ways. In this section, we briefly survey the kinds of extensions that are in wide use, highlighting some of the architectural issues that arise in delivering this extensibility. These features appear in most of the commercial DBMSs today to varying degrees, and in the open-source PostgreSQL DBMS as well.

4.7.1 Abstract Data Types

In principle, the relational model is agnostic to the choice of scalar data types that can be placed on schema columns. But the initial relational database systems supported only a static set of alphanumeric column types, and this limitation came to be associated with the rela-

tional model *per se*. A relational DBMS can be made extensible to new abstract data types at runtime, as was illustrated in the early Ingres-ADT system, and more aggressively in the follow-on Postgres system [88]. To achieve this, the DBMS type system — and hence the parser — has to be driven from the system catalog, which maintains the list of types known to the system, and pointers to the "methods" (code) used to manipulate the types. In this approach, the DBMS does not interpret types, it merely invokes their methods appropriately in expression evaluation; hence the name "abstract data types." As a typical example, one could register a type for 2-dimensional spatial "rectangles," and methods for operations like rectangle intersection or union. This also means that the system must provide a runtime engine for user-defined code, and safely execute that code without any risk of crashing the database server or corrupting data. All of today's major DBMSs allow users to define functions in an imperative "stored procedure" sublanguage of modern SQL. With the exception of MySQL, most support at least a few other languages, typically C and Java. On the Windows platform, Microsoft SQL Server and IBM DB2 support code compiled to the Microsoft. Net Common Language Runtime, which can be written in a wide variety of languages, most commonly Visual Basic, C++, and C#. PostgreSQL ships with support for C, Perl, Python and Tcl, and allows support for new languages to be added to the system at runtime — there are popular third-party plugins for Ruby and the open-source R statistical package.

To make abstract data types run efficiently in the DBMS, the query optimizer has to account for "expensive" user-defined code in selection and join predicates, and in some cases postpone selections until after joins [13, 37]. To make ADTs even more efficient, it is useful to be able to define indexes on them. At minimum, B+-trees need to be extended to index expressions over ADTs rather than just columns (sometimes termed "functional indexes"), and the optimizer has to be extended to choose them when applicable. For predicates other than linear orders ($<, >, =$), B+-trees are insufficient, and the system needs to support an extensible indexing scheme; two approaches in the literature are the original Postgres extensible access method interface [88], and the GiST [39].

4.7.2 Structured Types and XML

ADTs are designed to be fully compatible with the relational model — they do not in any way change the basic relational algebra, they only change the expressions over attribute values. Over the years, however, there have been many proposals for more aggressive changes to databases to support non-relational *structured* types: i.e., nested collection types like arrays, sets, trees, and nested tuples and/or relations. Perhaps the most relevant of these proposals today is the support for XML via languages like XPath and XQuery.[3] There are roughly three approaches to handling structured types like XML. The first is to build a custom database system that operates on data with structured types; historically these attempts have been overshadowed by approaches that accommodate the structured types within a traditional relational DBMS, and this trend was followed in the case of XML as well. The second approach is to treat the complex type as an ADT. For example, one can define a relational table with a column of type XML that stores an XML document per row. This means that expressions to search the XML — e.g., XPath tree-matching patterns — are executed in a manner that is opaque to the query optimizer. A third approach is for the DBMS to "normalize" the nested structure into a set of relations upon insertion, with foreign keys connecting subobjects to their parents. This technique, sometimes called "shredding" the XML, exposes all the structure of the data to the DBMS within a relational framework, but adds storage overhead, and requires joins to "reconnect" the data upon querying. Most DBMS vendors today offer both ADT and shredded options for storage, and allow the database designer to choose between them. In the case of XML, it is also common in the shredded approach to offer the option of removing the ordering information between XML elements nested at the same level, which

[3] XML is sometimes referred to as "semi-structured" data, because it places no restrictions on the structure of documents. However, unlike free-text, it *encourages* structure, including non-relational complexities like ordering and nested collections. The complications in storing and querying XML in databases tend to arise from handling the structure of an XML document, rather than from issues in dealing with unstructured text within the document. Indeed, many of the techniques used for query processing over XML have their roots in research on the richly structured ODMG object database model, and its OQL query language.

can improve query performance by allowing join reordering and other relational optimizations.

A related issue is more modest extensions to the relational model to handle nested tables and tuples, as well as arrays. These are widely used in Oracle installations, for example. The design trade-offs are similar in many ways to those of handling XML.

4.7.3 Full-Text Search

Traditionally, relational databases were notoriously poor at handling rich textual data and the keyword search that usually goes with it. In principle, modeling free-text in a database is a simple matter of storing the documents, defining an "inverted file" relation with tuples of the form `(word, documentID, position)`, and building a B+-tree index over the `word` column. This is roughly what happens in any text search engine, modulo some linguistic canonicalization of words, and some additional per-tuple attributes to aid in rank-ordering search results. But in addition to the model, most text indexing engines implement a number of performance optimizations specific to this schema, which are not implemented in a typical DBMS. These include "denormalizing" the schema to have each word appear only once with a list of occurrences per word, i.e., `(word, list <documentID, position>)` This allows for aggressive delta-compression of the list (typically called a "postings list"), which is critical given the characteristically skewed (Zipfian) distribution of words in documents. Moreover, text databases tend to be used in a data warehousing style, bypassing any DBMS logic for transactions. The common belief is that a naïve implementation of text search in a DBMS like the one above runs roughly an order of magnitude slower than the custom text indexing engines.

However, most DBMSs today either contain a subsystem for text indexing, or can be bundled with a separate engine to do the job. The text indexing facility can typically be used both over full-text documents, and over short textual attributes in tuples. In most cases the full-text index is updated asynchronously ("crawled") rather than being maintained transactionally; PostgreSQL is unusual in offering the option of a full-text index with transactional updates. In some systems,

the full-text index is stored outside the DBMS, and hence requires separate tools for backup and restore. A key challenge in handling full-text search in a relational database is to bridge the semantics of relational queries (unordered and complete sets of results) with ranked document search using keywords (ordered and typically incomplete results) in a way that is useful and flexible. For example, it is unclear how to order the output of a join query over two relations when there is a keyword search predicate on each relation. This issue remains rather *ad hoc* in current practice. Given a semantics for query output, another challenge is for the relational query optimizer to reason about selectivity and cost estimation for the text index, as well as judging the appropriate cost model for a query whose answer set is ordered and paginated in the user interface, and likely not to be retrieved completely. By all reports, this last topic is being aggressively pursued in a number of the popular DBMSs.

4.7.4 Additional Extensibility Issues

In addition to the three driving usage scenarios for database extensibility, we bring up two core components within the engine that are often made extensible for a variety of uses.

There have been a number of proposals for extensible query optimizers, including the designs that underpin the IBM DB2 optimizer [54, 68], and the designs that underpin the Tandem and Microsoft optimizers [25]. All these schemes provide rule-driven subsystems that generate or modify query plans, and allow new optimization rules to be registered independently. These techniques are helpful in making it easier to extend the optimizer when new features are added to the query executor, or when new ideas are developed for specific query rewrites or plan optimizations. These generic architectures were important in enabling many of the specific extensible type functionalities described above.

Another cross-cutting form of extensibility that arose since the early systems is the ability for the database to "wrap" remote data sources within the schema as if they were native tables, and access them during query processing. One challenge in this regard is for the optimizer to handle data sources that do not support scans, but will respond

requests that assign values to variables; this requires generalizing the optimizer logic that matches index SARGs to query predicates [33]. Another challenge is for the executor to deal efficiently with remote data sources that may be slow or bursty in producing outputs; this generalizes the design challenge of having the query executor do asynchronous disk I/O, increasing the access-time variability by an order of magnitude or more [22, 92].

4.8 Standard Practice

The coarse architecture of essentially all relational database query engines looks similar to that of the System R prototype [3]. Query processing research and development over the years has focused on innovations within that framework, to accelerate more and more classes of queries and schemas. Major design differences across systems arise in the optimizer search strategy (top-down vs. bottom-up), and in the query executor control-flow model, especially for shared-nothing and shared-disk parallelism (iterators and the exchange operator vs. asynchronous producer/consumer schemes). At a finer-grained level, there is a good deal of differentiation on the mix of schemes used within the optimizer, executor, and access methods to achieve good performance for different workloads including OLTP, decision-support for warehousing, and OLAP. This "secret sauce" within the commercial products determines how well they perform in specific circumstances, but to a first approximation all of the commercial systems do quite well across a broad range of workloads, and can be made to look slow in specific workloads.

In the open source arena, PostgreSQL has a reasonably sophisticated query processor with a traditional cost-based optimizer, a broad set of execution algorithms, and a number of extensibility features not found in commercial products. MySQL's query processor is far simpler, built around nested-loop joins over indices. The MySQL query optimizer focuses on analyzing queries to make sure that common operations are lightweight and efficient — particularly key/foreign-key joins, outer-join-to-join rewrites, and queries that ask for only the first few rows of the result set. It is instructive to read through the MySQL

manual and query processing code and compare it to the more involved traditional designs, keeping in mind the high adoption rate of MySQL in practice, and the tasks at which it seems to excel.

4.9 Discussion and Additional Material

Because of the clean modularity of query optimization and execution, there have been a huge number of algorithms, techniques, and tricks that have been developed in this environment over the years, and relational query processing research continues today. Happily, most of the ideas that have been used in practice (and many that have not) are found in the research literature. A good starting point for query optimization research is Chaudhuri's short survey [10]. For query processing research, Graefe offers a very comprehensive survey [24].

Beyond traditional query processing, there has been a great deal of work in recent years to incorporate rich statistical methods into the processing of large data sets. One natural extension is to use sampling or summary statistics to provide numerical approximations to aggregation queries [20], possibly in a continuously improving online fashion [38]. However, this has seen relatively slow uptake in the marketplace, despite fairly mature research results. Oracle and DB2 both provide simple base-table sampling techniques, but do not provide statistically robust estimation of queries involving more than one table. Instead of focusing on these features, most vendors have chosen to enrich their OLAP features instead, which constrain the family of queries that can be answered quickly, but provide users with 100% correct answers.

Another important but more fundamental extension has been to include "data mining" techniques in the DBMS. Popular techniques include statistical clustering, classification, regression, and association rules [14]. In addition to the standalone implementation of these techniques studied in the research literature, there are architectural challenges in integrating these techniques with rich relational queries [77].

Finally, it is worth noting that the broader computing community has recently become excited by data parallelism, as embodied by frameworks like Google's Map-Reduce, Microsoft's Dryad, and the open-

source Hadoop code that is supported by Yahoo! These systems are very much like shared-nothing-parallel relational query executors, with custom query operators implemented by the programmers of the application logic. They also include simple but sensibly engineered approaches to managing the failure of participating nodes, which is a common occurrence at large scales. Perhaps the most interesting aspect of this trend is the way that it is being creatively used for a variety of data-intensive problems in computing, including text and image processing, and statistical methods. It will be interesting to see whether other ideas from database engines get borrowed by users of these frameworks — for instance, there is early work at Yahoo! to extend Hadoop with declarative queries and an optimizer. Innovations built on these frameworks could be incorporated back into database engines as well.

5

Storage Management

Two basic types of DBMS storage managers are in commercial use today: either (1) the DBMS interacts directly with the low-level block-mode device drivers for the disks (often called raw-mode access), or (2) the DBMS uses standard OS file system facilities. This decision affects the DBMS's ability to control storage in both space and time. We consider these two dimensions in turn, and proceed to discuss the use of the storage hierarchy in more detail.

5.1 Spatial Control

Sequential bandwidth to and from disk is between 10 and 100 times faster than random access, and this ratio is increasing. Disk density has been doubling every 18 months and bandwidth rises approximately as the square root of density (and linearly with rotation speed). Disk arm movement, however, is improving at a much slower rate — about 7%/year [67]. As a result, it is critical for the DBMS storage manager to place blocks on the disk such that queries that require large amounts of data can access it sequentially. Since the DBMS can understand its workload access patterns more deeply than the underlying OS, it

makes sense for DBMS architects to exercise full control over the spatial positioning of database blocks on disk.

The best way for the DBMS to control spatial locality of its data is to store the data directly to the "raw" disk device and avoid the file system entirely. This works because raw device addresses typically correspond closely to physical proximity of storage locations. Most commercial database systems offer this functionality for peak performance. This technique, although effective, does have some drawbacks. First, it requires the DBA to devote entire disk partitions to the DBMS, which makes them unavailable to utilities (backups, etc.) that need a filesystem interface. Second, "raw disk" access interfaces are often OS-specific, which can make the DBMS more difficult to port. This is a hurdle, however, that most commercial DBMS vendors overcame years ago. Finally, developments in the storage industry like RAID, Storage Area Networks (SAN), and logical volume managers have become popular. We are now at a point where "virtual" disk devices are the norm in most scenarios today — the "raw" device interface is actually being intercepted by appliances or software that reposition data aggressively across one or more physical disks. As a result, the benefits of explicit physical control by the DBMS have been diluted over time. We discuss this issue further in Section 7.3.

An alternative to raw disk access is for the DBMS to create a very large file in the OS file system, and to manage positioning of data as offsets in that file. The file is essentially treated as a linear array of disk-resident pages. This avoids some of the disadvantages of raw device access and still offers reasonably good performance. In most popular file systems, if you allocate a very large file on an empty disk, the offsets in that file will correspond fairly closely to physical proximity of storage regions. Hence this is a good approximation to raw disk access, without the need to go directly to the raw device interface. Most virtualized storage systems are also designed to place close offsets in a file in nearby physical locations. Hence the relative control lost when using large files rather than raw disks is becoming less significant over time. Using the file system interface has other ramifications regarding temporal control, which we discuss in the next subsection.

As a data point, we recently compared direct raw access with large file access on a mid-sized system using one of the major commercial DBMSs and found only a 6% degradation when running the TPC-C benchmark [91], and almost no negative impact on less I/O-intensive workloads. DB2 reports file system overhead as low as 1% when using Direct I/O (DIO) and its variants such as concurrent I/O (CIO). Consequently, DBMS vendors typically no longer recommend raw storage, and few customers run in this configuration. It remains a supported feature in major commercial systems primarily for benchmark use.

Some commercial DBMS also allow the database page size to be custom-set to a size appropriate for the expected work load. Both IBM DB2 and Oracle support this option. Other commercial systems such as Microsoft SQL Server do not support multiple page sizes due to the increased administrative complexity this brings. If tunable page sizes are supported, the selected size should be a multiple of the page size used by the file system (or raw device if raw I/O is being used). A discussion of the appropriate choice of page sizes is given in the "5-minute rule" paper which has been subsequently updated to the "30-minute rule" [27]. If the file system is being used rather than raw device access, special interfaces may be required to write pages of a different size than that of the file system; the POSIX *mmap/msync* calls, for example, provide such support.

5.2 Temporal Control: Buffering

In addition to controlling *where* on the disk data should be placed, a DBMS must control *when* data gets physically written to the disk. As we will discuss in Section 5, a DBMS contains critical logic that reasons about when to write blocks to disk. Most OS file systems also provide built-in I/O buffering mechanisms to decide when to do reads and writes of file blocks. If the DBMS uses standard file system interfaces for writing, the OS buffering can confound the intention of the DBMS logic by silently postponing or reordering writes. This can cause major problems for the DBMS.

The first set of problems regard the *correctness* of the database's ACID transaction promise: the DBMS cannot guarantee atomic recov-

ery after software or hardware failure without explicitly controlling the timing and ordering of disk writes. As we will discuss in Section 5.3, the *write ahead logging* protocol requires that writes to the log device must precede corresponding writes to the database device, and commit requests cannot return to users until commit log records have been reliably written to the log device.

The second set of problems with OS buffering concern performance, but have no implications on correctness. Modern OS file systems typically have some built-in support for *read-ahead* (speculative reads) and *write-behind* (delayed, batched writes). These are often poorly suited to DBMS access patterns. File system logic depends on the contiguity of *physical* byte offsets in files to make read ahead decisions. DBMS-level I/O facilities can support *logical* predictive I/O decisions based upon future read requests that are known at the SQL query processing level but are not easily discernable at the file system level. For example, logical DBMS-level read-ahead can be requested when scanning the leaves of a B+-tree (rows are stored in the leaves of a B+-tree) that are not necessarily contiguous. Logical read-ahead is easily achieved in DBMS logic by having the DBMS issue I/O requests in advance of its needs. The query execution plan contains the relevant information about data access algorithms, and has full information about future access patterns for the query. Similarly, the DBMS may want to make its own decisions about when to flush the log tail, based on considerations that mix issues like lock contention with I/O throughput. This detailed future access pattern knowledge is available to the DBMS, but not to the OS file system.

The final performance issues are "double buffering" and the high CPU overhead of memory copies.Given that the DBMS has to do its own buffering carefully for correctness, any additional buffering by the OS is redundant. This redundancy results in two costs. First, it wastes system memory by effectively reducing the memory available for doing useful work. Second, it wastes time and processing resources, by causing an additional copying step: on reads, data is first copied from the disk to the OS buffer, and then copied again to the DBMS buffer pool. On writes, both of these copies are required in reverse.

Copying data in memory can be a serious bottleneck. Copies contribute latency, consume CPU cycles, and can flood the CPU data

cache. This fact is often a surprise to people who have not operated or implemented a database system, and assume that main-memory operations are "free" compared to disk I/O. But in practice, *throughput in a well-tuned transaction processing DBMS is typically not I/O-bound*. This is achieved in high-end installations by purchasing sufficient disks and RAM so that repeated page requests are absorbed by the buffer pool, and disk I/Os are shared across the disk arms at a rate that can feed the data appetite of all the processors in the system. Once this kind of "system balance" is achieved, I/O latencies cease to be the primary system throughput bottleneck, and the remaining main-memory bottlenecks become the limiting factors in the system. Memory copies are becoming a dominant bottleneck in computer architectures: this is due to the gap in performance evolution between raw CPU cycles per second per dollar (which follows Moore's law) and RAM access speed (which trails Moore's law significantly) [67].

The problems of OS buffering have been well known in the database research literature [86] and the industry for some time. Most modern OSs now provide hooks (e.g., the POSIX *mmap suite* calls or the platform specific DIO and CIO API sets) so that programs such as database servers can circumvent double-buffering the file cache. This ensures that writes go through to disk when requested, that double buffering is avoided, and that the DBMS can control the page replacement strategy.

5.3 Buffer Management

In order to provide efficient access to database pages, every DBMS implements a large shared buffer pool in its own memory space. In the early days the buffer pool was statically allocated to an administratively chosen value, but most commercial DBMSs now dynamically adjust the buffer pool size based upon system need and available resources. The buffer pool is organized as an array of *frames*, where each frame is a region of memory the size of a database disk block. Blocks are copied into the buffer pool from disk without format change, manipulated in memory in this native format, and later written back. This translation-free approach avoids CPU bottlenecks in "marshalling" and "unmarshalling" data to/from disk; perhaps more importantly,

fixed-sized frames sidestep the memory-management complexities of external fragmentation and compaction that generic techniques cause.

Associated with the array of buffer pool frames is a hash table that maps (1) page numbers currently held in memory to their location in the frame table, (2) the location for that page on backing disk storage, and (3) some metadata about the page. The metadata includes a *dirty bit* to indicate whether the page has changed since it was read from disk, and any information needed by the *page replacement policy* to choose pages to evict when the buffer pool is full. Most systems also include a *pin count* to signal that the page is not eligible for participation in the page-replacement algorithm. When the pin count is non-zero, the page is "pinned" in memory and will not be forced to disk or stolen. This allows the DBMS's worker threads to pin pages in the buffer pool by incrementing the pin count before manipulating the page, and then decrementing it thereafter. The intent is to have only a tiny fraction of the buffer pool pinned at any fixed point of time. Some systems also offer the ability to pin tables in memory as an administrative option, which can improve access times to small, heavily used tables. However, pinned pages reduce the number of pages available for normal buffer pool activities and can negatively impact performance as the percentage of pinned pages increases.

Much research in the early days of relational systems focused on the design of page replacement policies, since the diversity of data access patterns found in DBMSs render simple techniques ineffective. For example, certain database operations tend to require full table scans and, when the scanned table is much larger than the buffer pool, these operations tend to clear the pool of all commonly referenced data. For such access patterns, the recency of reference is a poor predictor of the probability of future reference, so OS page replacement schemes like LRU and CLOCK were well-known to perform poorly for many database access patterns [86]. A variety of alternative schemes were proposed, including some that attempted to tune the replacement strategy via query execution plan information [15]. Today, most systems use simple enhancements to LRU schemes to account for the case of full table scans. One that appears in the research literature and has been implemented in commercial systems is LRU-2 [64]. Another scheme

used in commercial systems is to have the replacement policy depend on the page type: e.g., the root of a B+-tree might be replaced with a different strategy than a page in a heap file. This is reminiscent of Reiter's Domain Separation scheme [15, 75].

Recent hardware trends, including 64-bit addressing and falling memory prices, have made very large buffer pools economically possible. This opens up new opportunities for exploiting large main memory for efficiency. As a counterpoint, a large and very active buffer pool also brings more challenges in restart recovery speed and efficient checkpointing, among other issues. These topics will be discussed further in Section 6.

5.4 Standard Practice

In the last decade, commercial file systems have evolved to the point where they can support database storage systems quite well. In the standard usage model, the system administrator creates a file system on each disk or logical volume in the DBMS. The DBMS then allocates a single large file in each of these file systems and controls placement of data within that file via low-level interfaces like the mmap suite. The DBMS essentially treats each disk or logical volume as a linear array of (nearly) contiguous database pages. In this configuration, modern file systems offer reasonable spatial and temporal control to the DBMS and this storage model is available in essentially all database system implementations. The raw disk support remains a common high-performance option in most database systems, however, its usage is rapidly narrowing to performance benchmarks only.

5.5 Discussion and Additional Material

Database storage subsystems are a very mature technology, but a number of new considerations have emerged for database storage in recent years, which have the potential to change data management techniques in a number of ways.

One key technological change is the emergence of flash memory as an economically viable random-access persistent storage technology [28]. Ever since the early days of database system research, there has been

discussion of sea-changes in DBMS design arising from new storage technologies replacing disk. Flash memory appears to be both technologically viable and economically supported by a broad market, and presents an interesting intermediate cost/performance trade-off relative to disk and RAM. Flash is the first new persistent storage medium to succeed in this regard in more than three decades, and hence its particulars may have significant impact on future DBMS designs.

Another traditional topic that has recently come to the fore is compression of database data. Early work on the topic focused on on-disk compression to minimize disk latencies during read, and maximize the capacity of the database buffer pool. As processor performance has improved and RAM latencies have not kept pace, it has become increasingly important to consider keeping data compressed even during computations, so as to maximize the residency of data in processor caches as well. But this requires compressed representations that are amenable to data processing, and query processing internals that manipulate compressed data. Another wrinkle in relational database compression is that databases are reorderable sets of tuples, whereas most work on compression focuses on byte streams without considering reordering. Recent research on this topic suggests significant promise for database compression in the near future [73].

Finally, outside the traditional relational database market there is enhanced interest in large-scale but sparse data storage techniques, where there are logically thousands of columns, the bulk of which are null for any given row. These scenarios are typically represented via some kind of set of attribute-value pairs or triples. Instances include Google's BigTable [9], the Tagged Columns used by Microsoft's Active Directory and Exchange products, and the Resource Description Framework (RDF) proposed for the "Semantic Web." Common to these approaches are the use of storage systems that organize disk in terms of the columns of data tables, rather than rows. The idea of column-oriented storage has been revived and explored in detail in a number of recent database research efforts [36, 89, 90].

6

Transactions: Concurrency Control and Recovery

Database systems are often accused of being enormous, monolithic software systems that cannot be split into reusable components. In practice, database systems — and the development teams that implement and maintain them — do break down into independent components with documented interfaces between them. This is particularly true of the interface between the relational query processor and the transactional storage engine. In most commercial systems these components are written by different teams and have well-defined interfaces between them.

The truly monolithic piece of a DBMS is the transactional storage manager that typically encompasses four deeply intertwined components:

1. A lock manager for concurrency control.
2. A log manager for recovery.
3. A buffer pool for staging database I/Os.
4. Access methods for organizing data on disk.

A great deal of ink has been spilled describing the fussy details of transactional storage algorithms and protocols in database systems. The reader wishing to become knowledgeable about these systems should read — at a minimum — a basic undergraduate database

textbook [72], the journal article on the ARIES log protocol [59], and at least one serious article on transactional index concurrency and logging [46, 58]. More advanced readers will want to consult Gray and Reuter's textbook on transactions [30]. To really become an expert, this reading has to be followed by an implementation effort. We do not dwell on algorithms and protocols here, but rather survey the roles of these various components. We focus on the system infrastructure that is often ignored in the textbooks, highlighting the inter-dependencies between components that lead to much of the subtlety and complexity in making the simple protocols workable.

6.1 A Note on ACID

Many people are familiar with the term "ACID transactions," a mnemonic due to Haerder and Reuter [34]. ACID stands for *Atomicity, Consistency, Isolation*, and *Durability*. These terms were not formally defined, and are not mathematical axioms that combine to guarantee transactional consistency. So it is not important to carefully distinguish the terms and their relationships. But despite the informal nature, the ACID acronym is useful to organize a discussion of transaction systems, and is sufficiently important that we review it here:

- *Atomicity* is the "all or nothing" guarantee for transactions — either all of a transaction's actions commit or none do.
- *Consistency* is an application-specific guarantee; SQL integrity constraints are typically used to capture these guarantees in a DBMS. Given a definition of consistency provided by a set of constraints, a transaction can only commit if it leaves the database in a consistent state.
- *Isolation* is a guarantee to application writers that two concurrent transactions will not see each other's in-flight (not-yet-committed) updates. As a result, applications need not be coded "defensively" to worry about the "dirty data" of other concurrent transactions; they can be coded as if the programmer had sole access to the database.

- *Durability* is a guarantee that the updates of a committed transaction will be visible in the database to subsequent transactions independent of subsequent hardware or software errors, until such time as they are overwritten by another committed transaction.

Roughly speaking, modern DBMSs implement isolation via a locking protocol. Durability is typically implemented via logging and recovery. Isolation and Atomicity are guaranteed by a combination of locking (to prevent visibility of transient database states), and logging (to ensure correctness of data that is visible). Consistency is managed by runtime checks in the query executor: if a transaction's actions will violate a SQL integrity constraint, the transaction is aborted and an error code returned.

6.2 A Brief Review of Serializability

We begin our discussion of transactions by briefly reviewing the main goal of database concurrency control, and proceed in the next section to describe two of the most important building blocks used to implement this notion in most multi-user transactional storage managers: (1) locking and (2) latching.

Serializability is the well-defined textbook notion of correctness for concurrent transactions. It dictates that a sequence of interleaved actions for multiple committing transactions must correspond to some serial execution of the transactions — as though there were no parallel execution at all. Serializability is a way of describing the desired behavior of a set of transactions. *Isolation* is the same idea from the point of view of a single transaction. A transaction is said to execute in isolation if it does not see any concurrency anomalies — the "I" of ACID.

Serializability is enforced by the DBMS concurrency control model. There are three broad techniques of concurrency control enforcement. These are well-described in textbooks and early survey papers [7], but we very briefly review them here:

1. *Strict two-phase locking (2PL)*: Transactions acquire a shared lock on every data record before reading it, and

an exclusive lock on every data item before writing it. All locks are held until the end of the transaction, at which time they are all released atomically. A transaction blocks on a wait-queue while waiting to acquire a lock.

2. *Multi-Version Concurrency Control* (*MVCC*): Transactions do not hold locks, but instead are guaranteed a consistent view of the database state at some time in the past, even if rows have changed since that fixed point in time.
3. *Optimistic Concurrency Control* (*OCC*): Multiple transactions are allowed to read and update an item without blocking. Instead, transactions maintain histories of their reads and writes, and before committing a transaction checks history for isolation conflicts they may have occurred; if any are found, one of the conflicting transactions is rolled back.

Most commercial relational DBMS implement full serializability via 2PL. The lock manager is the code module responsible for providing the facilities for 2PL.

In order to reduce locking and lock conflicts some DBMSs support MVCC or OCC, typically as an add-on to 2PL. In an MVCC model, read locks are not needed, but this is often implemented at the expense of not providing full serializability, as we will discuss shortly in Section 4.2.1. To avoid blocking writes behind reads, the write is allowed to proceed after the previous version of the row is either saved, or guaranteed to be quickly obtainable otherwise. The in-flight read transactions continue to use the previous row value as though it were locked and prevented from being changed. In commercial MVCC implementations, this stable read value is defined to be either the value at the start of the read transaction or the value at the start of that transaction's most recent SQL statement.

While OCC avoids waiting on locks, it can result in higher penalties during true conflicts between transactions. In dealing with conflicts across transactions, OCC is like 2PL except that it converts what would be lock-waits in 2PL into transaction rollbacks. In scenarios where conflicts are uncommon, OCC performs very well, avoiding overly conservative wait time. With frequent conflicts, however, exces-

sive rollbacks and retries negatively impact performance and make it a poor choice [2].

6.3 Locking and Latching

Database locks are simply names used by convention within the system to represent either physical items (e.g., disk pages) or logical items (e.g., tuples, files, volumes) that the DBMS manages. Note that any name can have a lock associated with it — even if that name represents an abstract concept. The locking mechanism simply provides a place to register and check for these names. Every lock is associated with a transaction and each transaction has a unique transaction ID. Locks come in different lock "modes," and these modes are associated with a lock-mode compatibility table. In most systems, this logic is based on the well-known lock modes that are introduced in Gray's paper on granularity of locks [29]. That paper also explains how hierarchical locks are implemented in commercial systems. Hierarchical locking allows a single lock to be used to lock an entire table and, at the same time, support row granularity locks in the same table both efficiently and correctly.

The lock manager supports two basic calls; `lock (lockname, transactionID, mode)`, and `remove_transaction (transaction-ID)`. Note that because of the strict 2PL protocol, there should not be an individual call to unlock resources individually — the `remove_transaction()` call will unlock all resources associated with a transaction. However, as we discuss in Section 5.2.1, the SQL standard allows for lower degrees of transaction isolation, and hence there is a need for an `unlock (lockname, transactionID)`call as well. There is also a `lock_upgrade (lockname, transactionID, newmode)` call to allow transactions to "upgrade" to higher lock modes (e.g., from shared to exclusive mode) in a two-phase manner, without dropping and re-acquiring locks. Additionally, some systems also support a `conditional_lock (lockname, transactionID, mode)` call. The `conditional_lock()` call always returns immediately, and indicates whether it succeeded in acquiring the lock. If it did not succeed, the calling DBMS thread is *not* enqueued waiting for the lock. The use of conditional locks for index concurrency is discussed in [60].

To support these calls, the lock manager maintains two data structures. A global *lock table* is maintained to hold lock names and their associated information. The lock table is a dynamic hash table keyed by (a hash function of) lock names. Associated with each lock is a *mode* flag to indicate the lock mode, and a *wait queue* of lock request pairs (transactionID, mode). In addition, the lock manager maintains a *transaction table* keyed by transactionID, which contains two items for each transaction T: (1) a pointer to T's DBMS thread state, to allow T's DBMS thread to be rescheduled when it acquires any locks it is waiting on, and (2) a list of pointers to all of T's lock requests in the lock table, to facilitate the removal of all locks associated with a particular transaction (e.g., upon transaction commit or abort).

Internally, the lock manager makes use of a *deadlock detector* DBMS thread that periodically examines the lock table to detect *waits-for cycles* (a cycle of DBMS workers where each is waiting for the next and a cycle is formed). Upon detection of a deadlock, the deadlock detector aborts one of the deadlocked transactions. The decision of which deadlocked transaction to abort is based on heuristics that have been studied in the research literature [76]. In shared-nothing and shared-disk systems, either distributed deadlock detection [61] or a more primitive timeout-based deadlock detector is required. A more detailed description of a lock manager implementation is given in Gray and Reuter's text [30].

As an auxiliary to database locks, lighter-weight *latches* are also provided for mutual exclusion. Latches are more akin to monitors [41] or semaphores than locks; they are used to provide exclusive access to internal DBMS data structures. As an example, the buffer pool page table has a latch associated with each frame, to guarantee that only one DBMS thread is replacing a given frame at any time. Latches are used in the implementation of locks and to briefly stabilize internal data structures potentially being concurrently modified.

Latches differ from locks in a number of ways:

- Locks are kept in the lock table and located via hash tables; latches reside in memory near the resources they protect, and are accessed via direct addressing.

- In a strict 2PL implementation, locks are subject to the strict 2PL protocol. Latches may be acquired or dropped during a transaction based on special-case internal logic.
- Lock acquisition is entirely driven by data access, and hence the order and lifetime of lock acquisitions is largely in the hands of applications and the query optimizer. Latches are acquired by specialized code inside the DBMS, and the DBMS internal code issues latch requests and releases strategically.
- Locks are allowed to produce deadlock, and lock deadlocks are *detected* and resolved via transactional restart. Latch deadlock must be *avoided*; the occurrence of a latch deadlock represents a bug in the DBMS code.
- Latches are implemented using an atomic hardware instruction or, in rare cases, where this is not available, via mutual exclusion in the OS kernel.
- Latch calls take at most a few dozen CPU cycles whereas lock requests take hundreds of CPU cycles.
- The lock manager tracks all the locks held by a transaction and automatically releases the locks in case the transaction throws an exception, but internal DBMS routines that manipulate latches must carefully track them and include manual cleanup as part of their exception handling.
- Latches are not tracked and so cannot be automatically released if the task faults.

The latch API supports the routines `latch(object, mode)`, `unlatch(object)`, and `conditional_latch(object, mode)`. In most DBMSs, the choices of latch modes include only shared or exclusive. Latches maintain a mode, and a waitqueue of DBMS threads waiting on the latch. The latch and unlatch calls work as one might expect. The `conditional_latch()` call is analogous to the `conditional_lock()` call described above, and is also used for index concurrency [60].

6.3.1 Transaction Isolation Levels

Very early in the development of the transaction concept, attempts were made to increase concurrency by providing "weaker" semantics

than serializability. The challenge was to provide robust definitions of the semantics in these cases. The most influential effort in this regard was Gray's early work on "Degrees of Consistency" [29]. That work attempted to provide both a declarative definition of consistency degrees, and implementations in terms of locking. Influenced by this work, the ANSI SQL standard defines four "Isolation Levels":

1. *READ UNCOMMITTED*: A transaction may read any version of data, committed or not. This is achieved in a locking implementation by read requests proceeding without acquiring any locks.[1]
2. *READ COMMITTED*: A transaction may read *any committed* version of data. Repeated reads of an object may result in different (committed) versions. This is achieved by read requests acquiring a read lock before accessing an object, and unlocking it immediately after access.
3. *REPEATABLE READ*: A transaction will read only one version of committed data; once the transaction reads an object, it will always read the same version of that object. This is achieved by read requests acquiring a read lock before accessing an object, and holding the lock until end-of-transaction.
4. *SERIALIZABLE*: Full serializable access is guaranteed.

At first blush, REPEATABLE READ seems to provide full serializability, but this is not the case. Early in the System R project [3], a problem arose that was dubbed the "phantom problem." In the phantom problem, a transaction accesses a relation more than once with the same predicate in the same transaction, but sees new "phantom" tuples on re-access that were not seen on the first access. This is because two-phase locking at tuple-level granularity does not prevent the insertion of new tuples into a table. Two-phase locking of tables prevents phantoms, but table-level locking can be restrictive in cases

[1] In all isolation levels, write requests are preceded by write locks that are held until end of transaction.

where transactions access only a few tuples via an index. We investigate this issue further in Section 5.4.3 when we discuss locking in indexes.

Commercial systems provide the four isolation levels above via locking-based implementations of concurrency control. Unfortunately, as noted by Berenson et al. [6], neither the early work by Gray nor the ANSI standard achieve the goal of providing truly declarative definitions. Both rely in subtle ways on an assumption that a locking scheme is used for concurrency control, as opposed to an optimistic [47] or multi-version [74] concurrency scheme. This implies that the proposed semantics are ill-defined. The interested reader is encouraged to look at the Berenson paper which discusses some of the problems in the SQL standard specifications, as well as the research by Adya et al. [1], which provides a new, cleaner approach to the problem.

In addition to the standard ANSI SQL isolation levels, various vendors provide additional levels that have proven popular in particular cases.

- *CURSOR STABILITY*: This level is intended to solve the "lost update" problem of READ COMMITTED. Consider two transactions T1 and T2. T1 runs in READ COMMITTED mode, reads an object X (say the value of a bank account), remembers its value, and subsequently writes object X based on the remembered value (say adding $100 to the original account value). T2 reads and writes X as well (say subtracting $300 from the account). If T2's actions happen between T1's read and T1's write, then the effect of T2's update will be lost — the final value of the account in our example will be up by $100, instead of being down by $200 as desired. A transaction in CURSOR STABILITY mode holds a lock on the most recently read item on a query cursor; the lock is automatically dropped when the cursor is moved (e.g., via another FETCH) or the transaction terminates. CURSOR STABILITY allows the transaction to do read–think–write sequences on individual items without intervening updates from other transactions.

- *SNAPSHOT ISOLATION*: A transaction running in SNAPSHOT ISOLATION mode operates on a version of the database as it existed at the time the transaction began; subsequent updates by other transactions are invisible to the transaction. This is one of the major uses of MVCC in production database systems. When the transaction starts, it gets a unique *start-timestamp* from a monotonically increasing counter; when it commits it gets a unique *end-timestamp* from the counter. The transaction commits only if no other transaction with an overlapping *start/end-transaction* pair wrote data that this transaction also wrote. This isolation mode depends upon a multi-version concurrency implementation, rather than locking. These schemes typically coexist, however, in systems that support SNAPSHOT ISOLATION.
- *READ CONSISTENCY*: This is an MVCC scheme defined by Oracle; it is subtly different from SNAPSHOT ISOLATION. In the Oracle scheme, each SQL *statement* (of which there may be many in a single transaction) sees the most recently committed values as of the start of the statement. For statements that fetch from cursors, the cursor set is based on the values as of the time it is opened. This is implemented by maintaining multiple logical versions of individual tuples, with a single transaction possibly referencing multiple versions of a single tuple. Rather than storing each version that might be needed, Oracle stores only the most recent version. If an older version is needed, it produces the older version by taking the current version and "rolling it back" by applying undo log records as needed. Modifications are maintained via long-term write locks, so when two transactions want to write the same object, the first writer "wins" and the second writer must wait for the transaction completion of the first writer before its write proceeds. By contrast, in SNAPSHOT ISOLATION the first committer "wins" rather than the first writer.

Weak isolation schemes can provide higher concurrency than full serializability. As a result, some systems even use weak consistency

as the default. For example, Microsoft SQL Server defaults to READ COMMITTED. The downside is that Isolation (in the ACID sense) is not guaranteed. Hence application writers need to reason about the subtleties of the schemes to ensure that their transactions run correctly. This can be tricky given the operationally defined semantics of the schemes, and can lead to applications being more difficult to move between DBMSs.

6.4 Log Manager

The log manager is responsible for maintaining the durability of committed transactions, for facilitating the rollback of aborted transactions to ensure atomicity, and for recovering from system failure or non-orderly shutdown. To provide these features, the log manager maintains a sequence of log records on disk, and a set of data structures in memory. In order to support correct behavior after a crash, the memory-resident data structures obviously need to be re-creatable from persistent data in the log and the database.

Database logging is an extremely complex and detail-oriented topic. The canonical reference on database logging is the journal paper on ARIES [59], and a database expert should be familiar with the details of that paper. The ARIES paper not only explains logging protocol, but also provides discussion of alternative design possibilities, and the problems that they can cause. This makes for dense, but ultimately rewarding reading. As a more digestible introduction, the Ramakrishnan and Gehrke textbook [72] provides a description of the basic ARIES protocol without side discussions or refinements. Here we discuss some of the basic ideas in recovery, and try to explain the complexity gap between textbook and journal descriptions.

The standard theme of database recovery is to use a Write-Ahead Logging (WAL) protocol. The WAL protocol consists of three very simple rules:

1. Each modification to a database page should generate a log record, and the log record must be flushed to the log device *before* the database page is flushed.

2. Database log records must be flushed in order; log record r cannot be flushed until all log records preceding r are flushed.
3. Upon a transaction commit request, a commit log record must be flushed to the log device *before* the commit request returns successfully.

Many people only remember the first of these rules, but all three are required for correct behavior.

The first rule ensures that the actions of incomplete transactions can be *undone* in the event of a transaction abort, to ensure atomicity. The combination of rules (2) and (3) ensure durability: the actions of a committed transaction can be *redone* after a system crash if they are not yet reflected in the database.

Given these simple principles, it is surprising that efficient database logging is as subtle and detailed as it is. In practice, however, the simple story above is complicated by the need for extreme performance. The challenge is to guarantee efficiency in the "fast path" for transactions that commit, while also providing high-performance rollback for aborted transactions, and quick recovery after crashes. Logging gets even more complex when application-specific optimizations are added, e.g., to support improved performance for fields that can only be incremented or decremented ("escrow transactions").

In order to maximize the speed of the fast path, most commercial database systems operate in a mode that Haerder and Reuter call "DIRECT, STEAL/NOT-FORCE" [34]: (a) data objects are updated in place, (b) unpinned buffer pool frames can be "stolen" (and the modified data pages written back to disk) even if they contain uncommitted data, and (c) buffer pool pages need *not* be "forced" (flushed) to the database before a commit request returns to the user. These policies keep the data in the location chosen by the DBA, and they give the buffer manager and disk scheduler full latitude to decide on memory management and I/O policies without consideration for transactional correctness. These features can have major performance benefits, but require that the log manager efficiently handle all the subtleties of *undoing* the flushes of stolen pages from aborted transactions, and *redoing* the changes to not-forced pages of committed transactions that are

lost on crash. One optimization used by some DBMSs is to combine the scalability advantages of a DIRECT, STEAL/NOT-FORCE system with the performance of a DIRECT NOT-STEAL/NOT-FORCE system. In these systems, pages are not stolen unless there are no clean pages remaining in the buffer pool, in which case the system degrades back to a STEAL policy with the additional overhead described above.

Another fast-path challenge in logging is to keep log records as small as possible, in order to increase the throughput of log I/O activity. A natural optimization is to log *logical* operations (e.g., "insert (Bob, $25000) into EMP") rather than physical operations (e.g., the after-images for all byte ranges modified via the tuple insertion, including bytes on both heap file and index blocks.) The trade-off is that the logic to redo and undo logical operations becomes quite involved. This can severely degrade performance during transaction abort and database recovery.[2] In practice, a mixture of physical and logical logging (so-called "physiological" logging) is used. In ARIES, physical logging is generally used to support REDO, and logical logging is used to support UNDO. This is part of the ARIES rule of "repeating history" during recovery to reach the crash state, and then rolling back transactions from that point.

Crash recovery is required to restore the database to a consistent state after a system failure or non-orderly shutdown. As explained above, recovery is theoretically achieved by replaying history and stepping through log records from the first all the way to the most recent record. This technique is correct, but not very efficient since the log could be arbitrarily long. Rather than starting from the very first log record, a correct result will be obtained by starting recovery at the oldest of these two log records: (1) the log record describing the earliest change to the oldest dirty page in the buffer pool, and (2) the log record representing the start of the oldest transaction in the system. The sequence number of this point is called the *recovery log sequence number* (recovery LSN). Since computing and recording the recovery LSN incurs overhead, and since we know that the recovery LSN is

[2] Note also that logical log records must always have well-known, inverse functions if they need to participate in undo processing.

monotonically increasing, we do not need to keep it always up to date. Instead we compute it at periodic intervals called *checkpoints*.

A naïve checkpoint would force all dirty buffer pool pages and then compute and store the recovery LSN. With a large buffer pool, this could lead to delays of several seconds to complete the I/O of the pending pages. So a more efficient "fuzzy" scheme for checkpointing is required, along with logic to correctly bring the checkpoint up to the most recent consistent state by processing as little of the log as possible. ARIES uses a very clever scheme in which the actual checkpoint records are quite tiny, containing just enough information to initiate the log analysis process and to enable the recreation of main-memory data structures lost at crash time. During an ARIES fuzzy checkpoint, the recovery LSN is computed but no buffer pool pages need to be synchronously written out. A separate policy is used to determine when to asynchronously write out old dirty buffer pool pages.

Note that rollback will require that log records be written. This can lead to difficult situations where the in-flight transactions cannot proceed due to running out of log space but they cannot be rolled back either. This situation is typically avoided through space reservation schemes, however, these schemes are hard to get and keep correct as the system evolves through multiple releases.

Finally, the task of logging and recovery is further complicated by the fact that a database is not merely a set of user data tuples on disk pages; it also includes a variety of "physical" information that allows it to manage its internal disk-based data structures. We discuss this in the context of index logging in the next section.

6.5 Locking and Logging in Indexes

Indexes are physical storage structures for accessing data in the database. The indexes themselves are invisible to database application developers, except inasmuch as they improve performance or enforce uniqueness constraints. Developers and application programs cannot directly observe or manipulate entries in indexes. This allows indexes to be managed via more efficient (and complex) transactional schemes. The only invariant that index concurrency and recovery needs to pre-

serve is that the index always returns transactionally consistent tuples from the database.

6.5.1 Latching in B+-Trees

A well-studied example of this issue arises in B+-tree latching. B+-trees consist of database disk pages that are accessed via the buffer pool, just like data pages. Hence one scheme for index concurrency control is to use two-phase locks on index pages. This means that every transaction that touches the index needs to lock the root of the B+-tree until commit time — a recipe for limited concurrency. A variety of latch-based schemes have been developed to work around this problem without setting any transactional locks on index pages. The key insight in these schemes is that modifications to the tree's *physical structure* (e.g., splitting pages) can be made in a non-transactional manner as long as all concurrent transactions continue to find the correct data at the leaves. There are roughly three approaches to this:

- *Conservative schemes*: Multiple transactions wanting to access the same pages are allowed only if they can be guaranteed not to conflict in their use of a page's content. One such conflict is a reading transaction that wants to traverse a fully packed internal page of the tree while a concurrent inserting transaction is operating below that page and might need to split that page [4]. These conservative schemes sacrifice too much concurrency compared with the more recent ideas below.
- *Latch-coupling schemes*: The tree traversal logic latches each node before it is visited, only unlatching a node when the next node to be visited has been successfully latched. This scheme is sometimes called latch "crabbing," because of the crablike movement of "holding" a node in the tree, "grabbing" its child, releasing the parent, and repeating. Latch coupling is used in some commercial systems; IBM's ARIES-IM version is well-described [60]. ARIES-IM includes some fairly intricate details and corner cases — on occasion it has to restart traversals after splits, and even set (very short-term) tree-wide latches.

- *Right-link schemes*: Simple additional structures are added to the B+-tree to minimize the requirement for latches and re-traversals. In particular, a link is added from each node to its right-hand neighbor. During traversal, right-link schemes do no latch-coupling — each node is latched, read, and unlatched. The main intuition in right-link schemes is that if a traversing transaction follows a pointer to a node n and finds that n was split in the interim, the traversing transaction can detect this fact, and "move right" via the right-links to find the new correct location in the tree [46, 50]. Some systems support reverse traversal using back-links as well.

Kornacker et al. [46] provide a detailed discussion of the distinctions between latch-coupling and right-link schemes, and point out that latch-coupling is only applicable to B+-trees, and will not work for index trees over more complex data, e.g., geographic data that does not have a single linear order. The PostgreSQL Generalized Search Tree (GiST) implementation is based on Kornacker et al.'s extensible right-link scheme.

6.5.2 Logging for Physical Structures

In addition to special-case concurrency logic, indexes also use special-case logging logic. This logic makes logging and recovery much more efficient, at the expense of increased code complexity. The main idea is that structural index changes need not be undone when the associated transaction is aborted; such changes can often have no effect on the database tuples seen by other transactions. For example, if a B+-tree page is split during an inserting transaction that subsequently aborts, there is no pressing need to undo the split during the abort processing.

This raises the challenge of labeling some log records *redo-only*. During any undo processing of the log, the redo-only changes can be left in place. ARIES provides an elegant mechanism for these scenarios, called *nested top actions*, that allows the recovery process to "jump over" log records for physical structure modifications during recovery without any special-case code.

This same idea is used in other contexts, including heap files. An insertion into a heap file may require that the file be extended on disk. To capture this, changes must be made to the file's *extent map*. This is a data structure on disk that points to the runs of contiguous blocks that constitute the file. These changes to the extent map need not be undone if the inserting transaction aborts. The fact that the file has become larger is a transactionally invisible side-effect, and may in fact be useful for absorbing future insert traffic.

6.5.3 Next-Key Locking: Physical Surrogates for Logical Properties

We close this section with a final index concurrency problem that illustrates a subtle but significant idea. The challenge is to provide full serializability (including phantom protection) while allowing for tuple-level locks and the use of indexes. Note that this technique only applies to full serializability and is not required or used in relaxed isolation models.

The phantom problem can arise when a transaction accesses tuples via an index. In such cases, the transaction typically does not lock the entire table, just the tuples in the table that are accessed via the index (e.g., "Name BETWEEN 'Bob' AND 'Bobby'"). In the absence of a table-level lock, other transactions are free to insert new tuples into the table (e.g., "Name='Bobbie'"). When these new inserts fall within the value-range of a query predicate, they will appear in subsequent accesses via that predicate. Note that the phantom problem relates to the visibility of database tuples, and hence is a problem with locks, not just latches. In principle, what is needed is the ability to somehow lock the *logical space* represented by the original query's search predicate, e.g., the range of all possible strings that fall between "Bob" and "Bobby" in lexicographic order. Unfortunately, predicate locking is expensive, since it requires a way to compare arbitrary predicates for overlap. This cannot be done with a hash-based lock table [3].

A common approach to the phantom problem in B+-trees is called *next-key locking*. In next-key locking, the index insertion code is modified so that an insertion of a tuple with index key $k$must allocate an

exclusive lock on the next-key tuple that exists in the index, where the next-key tuple has the lowest key greater than k. This protocol ensures that subsequent insertions cannot appear in between two tuples that were returned previously to an active transaction. It also ensures that tuples cannot be inserted just below the lowest-keyed tuple previously returned. If no "Bob" key was found on the first access, for example, then one should not be found on subsequent accesses within the same transaction. One case remains: the insertion of tuples just *above* the highest-keyed tuple previously returned. To protect against this case, the next-key locking protocol requires read transactions get a shared lock on the next-key tuple in the index as well. In this scenario, the next-key tuple is the minimum-keyed tuple that does *not* satisfy the query predicate. Updates behave logically as delete followed by insert although optimizations are both possible and common.

Next key locking, although effective, does suffer from over-locking and this can be problematic with some workloads. For example, if we were scanning records from key 1 through key 10, but the column being indexed had only the keys 1, 5, and 100 stored, the entire range from 1 to 100 would be read-locked since 100 is the next key after 10.

Next-key locking is not simply a clever hack. It is an example of using a physical object (a currently-stored tuple) as a *surrogate* for a logical concept (a predicate). The benefit is that simple system infrastructure like hash-based lock tables can be used for more complex purposes, simply by modifying the lock protocol. Designers of complex software systems should keep this general approach of logical surrogates in their "bag of tricks" when such semantic information is available.

6.6 Interdependencies of Transactional Storage

We claimed early in this section that transactional storage systems are monolithic, deeply entwined systems. In this section, we discuss a few of the interdependencies between the three main aspects of a transactional storage system: concurrency control, recovery management, and access methods. In a happier world, it would be possible to identify narrow APIs between these modules that would allow the implementations behind those APIs to be swappable. Our examples in this section show that this is not easily done. We do not intend to provide an exhaustive

list of interdependencies here; generating and proving the completeness of such a list would be a very challenging exercise. We do hope, however, to illustrate some of the twisty logic of transactional storage, and thereby justify the resulting monolithic implementations in commercial DBMSs.

We begin by considering concurrency control and recovery alone without the further complication of access methods. Even with this simplification, components are deeply intertwined. One manifestation of the relationship between concurrency and recovery is that write-ahead logging makes implicit assumptions about the locking protocol. Write-ahead logging requires *strict* two-phase locking, and will not operate correctly with non-strict two-phase locking. To see this, consider what happens during the rollback of an aborted transaction. The recovery code begins processing the log records of the aborted transaction, undoing its modifications. Typically this requires changing pages or tuples that the transaction previously modified. In order to make these changes, the transaction needs to have locks on those pages or tuples. In a non-strict 2PL scheme, if the transaction drops any locks before aborting, it may be unable to re-acquire the locks it needs to complete the rollback process.

Access methods complicate things yet further. It is a significant intellectual and engineering challenge to take a textbook access method algorithm (e.g., linear hashing [53] or R-trees [32]) and implement a correct, high-concurrency, recoverable version in a transactional system. For this reason, most leading DBMSs still only implement heap files and B+-trees as transactionally protected access methods; PostgreSQL's GiST implementation is a notable exception. As we illustrated above for B+-trees, high-performance implementations of transactional indexes include intricate protocols for latching, locking, and logging. The B+-trees in serious DBMSs are riddled with calls to the concurrency and recovery code. Even simple access methods like heap files have some tricky concurrency and recovery issues surrounding the data structures that describe their contents (e.g., extent maps). This logic is not generic to all access methods — it is very much customized to the specific logic of the access method and its particular implementation.

Concurrency control in access methods has been well-developed only for locking-oriented schemes. Other concurrency schemes (e.g.,

Optimistic or Multi-version concurrency control) do not usually consider access methods at all, or mention them only in an offhanded and impractical fashion [47]. Hence mixing and matching different concurrency mechanisms for a given access method implementation is difficult.

Recovery logic in access methods is particularly system-specific: the timing and contents of access method log records depend upon fine details of the recovery protocol, including the handling of structure modifications (e.g., whether they get undone upon transaction rollback, and if not how that is avoided), and the use of physical and logical logging. Even for a specific access method like a B+-tree, the recovery and concurrency logic are intertwined. In one direction, the recovery logic depends upon the concurrency protocol: if the recovery manager has to restore a physically consistent state of the tree, then it needs to know what inconsistent states could possibly arise, to bracket those states appropriately with log records for atomicity (e.g., via nested top actions). In the opposite direction, the concurrency protocol for an access method may be dependent on the recovery logic. For example, the right-link scheme for B+-trees assume that pages in the tree never "re-merge" after they split. This assumption requires that the recovery scheme use a mechanism such as nested top actions to avoid undoing splits generated by aborted transactions.

The one bright spot in this picture is that buffer management is relatively well-isolated from the rest of the components of the storage manager. As long as pages are pinned correctly, the buffer manager is free to encapsulate the rest of its logic and re-implement it as needed. For example, the buffer manager has freedom in the choice of pages to replace (because of the STEAL property), and the scheduling of page flushes (thanks to the NOT FORCE property). Achieving this isolation, of course, is the direct cause of much of the complexity in concurrency and recovery. So this spot is perhaps less bright than it seems.

6.7 Standard Practice

All production databases today support ACID transactions. As a rule, they use write-ahead logging for durability, and two-phase locking for

concurrency control. An exception is PostgreSQL, which uses multi-version concurrency control throughout. Oracle pioneered the limited use of multi-version concurrency side-by-side with locking as a way to offer relaxed consistency models like Snapshot Isolation and Read Consistency; the popularity of these modes among users has led to their adoption in more than one commercial DBMS, and in Oracle this is the default. B+-tree indexing are standard in all of the production databases, and most of the commercial database engines offer some form of multi-dimensional index either embedded in the system or as a "plugin" module. Only PostgreSQL offers high-concurrency multi-dimensional and text indexing, via its GiST implementation.

MySQL is unique in actively supporting a variety of storage managers underneath, to the point where DBAs often choose different storage engines for different tables in the same database. Its default storage engine, MyISAM, only supports table-level locking, but is considered the high-performance choice for read-mostly workloads. For read/write workloads, the InnoDB storage engine is recommended; it offers row-level locking. (InnoDB was purchased by Oracle some years ago, but remains open-source and free for use for the time being.) Neither of the MySQL storage engines provide the well-known hierarchical locking scheme developed for System R [29], despite its universal use in the other database systems. This makes the choice between InnoDB and MyISAM tricky for MySQL DBAs, and in some mixed-workload cases neither engine can provide good lock granularity, requiring the DBA to develop a physical design using multiple tables and/or database replication to support both scans and high-selectivity index access. MySQL also supports storage engines for main-memory and cluster-based storage, and some third-party vendors have announced MySQL-compliant storage engines, but most of the energy in the MySQL userbase today is focused on MyISAM and InnoDB.

6.8 Discussion and Additional Material

Transaction mechanisms are by now an extremely mature topic, and most of the possible tricks have been tried in one form or another over the years; new designs tend to involve permutations and combinations

of existing ideas. Perhaps the most noticeable changes in this space are due to the rapidly dropping price of RAM. This increases the motivation to keep a large fraction of the "hot" portions of the database in memory and run at memory speeds, which complicates the challenge of getting data flushed to persistent storage often enough to keep restart times low. The role of flash memory in transaction management is part of this evolving balancing act.

An interesting development in recent years is the relatively broad adoption of write-ahead logging in the OSs community, typically under the rubric of *Journaling* file systems. These have become standard options in essentially all OSs today. Since these filesystems typically still do not support transactions over file data, it is interesting to see how and where they use write-ahead logging for durability and atomicity. The interested reader is referred to [62, 71] for further reading. Another interesting direction in this regard is the work on *Stasis* [78], which attempts to better modularize ARIES-style logging and recovery and make it available to systems programmers for a wide variety of uses.

7

Shared Components

In this section, we cover a number of shared components and utilities that are present in nearly all commercial DBMS, but rarely discussed in the literature.

7.1 Catalog Manager

The database catalog holds information about data in the system and is a form of *metadata*. The catalog records the names of basic entities in the system (users, schemas, tables, columns, indexes, etc.) and their relationships, and is itself stored as a set of tables in the database. By keeping the metadata in the same format as the data, the system is made both more compact and simpler to use: users can employ the same language and tools to investigate the metadata that they use for other data, and the internal system code for managing the metadata is largely the same as the code for managing other tables. This code and language reuse is an important lesson that is often overlooked in early stage implementations, typically to the significant regret of developers later on. One of the authors witnessed this mistake yet again in an industrial setting within the last decade.

The basic catalog data is treated somewhat differently from normal tables for efficiency reasons. High-traffic portions of the catalog are often materialized in main memory as needed, typically in data structures that "denormalize" the flat relational structure of the catalogs into a main-memory network of objects. This lack of data independence in memory is acceptable because the in-memory data structures are used in a stylized fashion only by the query parser and optimizer. Additional catalog data is cached in query plans at parsing time, again often in a denormalized form suited to the query. Moreover, catalog tables are often subject to special-case transactional tricks to minimize "hot spots" in transaction processing.

Catalogs can become formidably large in commercial applications. One major Enterprise Resource Planning application, for example, has over 60,000 tables, with between 4 and 8 columns per table, and typically two or three indexes per table.

7.2 Memory Allocator

The textbook presentation of DBMS memory management tends to focus entirely on the buffer pool. In practice, database systems allocate significant amounts of memory for other tasks as well. The correct management of this memory is both a programming burden and a performance issue. Selinger-style query optimization can use a great deal of memory, for example, to build up state during dynamic programming. Query operators like hashjoins and sorts allocate significant memory at runtime. Memory allocation in commercial systems is made more efficient and easier to debug via the use of a *context-based* memory allocator.

A memory context is an in-memory data structure that maintains a list of *regions* of contiguous virtual memory, often called memory pools. Each region can have a small header that contains either a context label or a pointer to the context header structure. The basic API for memory contexts includes calls to:

- *Create a context with a given name or type.* The context type might advise the allocator how to efficiently handle memory allocation. For example, the contexts for the query optimizer

grow via small increments, while contexts for hashjoins allocate their memory in a few large batches. Based on such knowledge, the allocator can choose to allocate bigger or smaller regions at a time.

- *Allocate a chunk of memory within a context.* This allocation will return a pointer to memory (much like the traditional malloc () call). That memory may come from an existing region in the context. Or, if no such space exists in any region, the allocator will ask the OS for a new region of memory, label it, and link it into the context
- *Delete a chunk of memory within a context.* This may or may not cause the context to delete the corresponding region. Deletion from memory contexts is somewhat unusual. A more typical behavior is to delete an entire context.
- *Delete a context.* This first frees all of the regions associated with the context, and then deletes the context header.
- *Reset a context.* This retains the context, but returns it to the state of original creation, typically by deallocating all previously allocated regions of memory.

Memory contexts provide important software engineering advantages. The most important is that they serve as a lower-level, programmer-controllable alternative to garbage collection. For example, the developers writing the optimizer can allocate memory in an optimizer context for a particular query, without worrying about how to free the memory later on. When the optimizer has picked the best plan, it can copy the plan into memory from a separate executor context for the query, and then simply delete the query's optimizer context. This saves the trouble of writing code to carefully walk all the optimizer data structures and delete their components. It also avoids tricky memory leaks that can arise from bugs in such code. This feature is very useful for the naturally "phased" behavior of query execution, where control proceeds from parser to optimizer to executor, with a number of allocations in each context followed by a context deletion.

Note that memory contexts actually provide more control than most garbage collectors as developers can control both *spatial* and *temporal*

locality of deallocation. The context mechanism itself provides the spatial control that allows the programmer to separate memory into logical units. Temporal control follows from programmers being allowed to issue context deletions when appropriate. By contrast, garbage collectors typically work on all of a program's memory, and make their own decisions about when to run. This is one of the frustrations of attempting to write server-quality code in Java [81].

Memory contexts also provide performance advantages on platforms where the overhead for `malloc()` and `free()` is relatively high. In particular, memory contexts can use semantic knowledge (via the context type) of how memory will be allocated and deallocated, and may call `malloc()` and `free()` accordingly to minimize OS overheads. Some components of a database system (e.g., the parser and optimizer) allocate a large number of small objects, and then free them all at once via a context deletion. Calls to free `()` many small objects are rather expensive on most platforms. A memory allocator can instead call `malloc()` to allocate large regions, and apportion the resulting memory to its callers. The relative lack of memory deallocations means that the compaction logic that `malloc()` and `free()` use is not needed. And when the context is deleted, only a few `free()` calls are required to remove the large regions.

The interested reader may want to browse the open-source PostgreSQL code. This utilizes a fairly sophisticated memory allocator.

7.2.1 A Note on Memory Allocation for Query Operators

Vendors differ philosophically in their memory-allocation schemes for space-intensive operators such as hash joins and sorts. Some systems (e.g., DB2 for zSeries) allow the DBA to control the amount of RAM that such operations will use, and guarantee that each query gets that amount of RAM when executed. The admission control policy ensures this guarantee. In such systems, operators allocate their memory from the heap via the memory allocator. These systems provide good performance stability, but force the DBA to (statically) decide how to balance physical memory across various subsystems such as the buffer pool and the query operators.

Other systems (e.g., MS SQL Server), take the memory allocation task out of the DBA's hands and manage these allocations automatically. These systems attempt to intelligently allocate memory across the various components of query execution, including page caching in the buffer pool and query operator memory use. The pool of memory used for all of these tasks is the buffer pool itself. Hence the query operators in these systems take memory from the buffer pool via a DBMS-implemented memory allocator and only use the OS allocator for contiguous requests larger than a buffer pool page.

This distinction echoes our discussion of query preparation in Section 6.3.1. The former class of systems assumes that the DBA is engaged in sophisticated tuning, and that the workload for the system will be amenable to carefully chosen adjustments to the system's memory "knobs." Under these conditions, such systems should always perform predictably well. The latter class assumes that DBAs either do not or cannot correctly set these knobs, and attempts to replace DBA tuning with software logic. They also retain the right to change their relative allocations adaptively. This provides the possibility for better performance on changing workloads. As discussed in Section 6.3.1, this distinction says something about how these vendors expect their products to be used, and about the administrative expertise (and financial resources) of their customers.

7.3 Disk Management Subsystems

DBMS textbooks tend to treat disks as homogeneous objects. In practice, disk drives are complex and heterogeneous pieces of hardware that vary widely in capacity and bandwidth. Hence every DBMS has a disk management subsystem that deals with these issues and manages the allocation of tables and other units of storage across raw devices, logical volumes or files.

One responsibility of this subsystem is to map tables to devices and/or files. One-to-one mappings of tables to files sounds natural, but raised significant problems in early file systems. First, OS files traditionally could not be larger than a disk, while database tables may need to span multiple disks. Second, allocating too many OS files was

considered bad form, since the OS typically only allowed a few open file descriptors, and many OS utilities for directory management and backup did not scale to very large numbers of files. Finally, many early file systems limited file size to 2 GB. This is clearly an unacceptably small table limit. Many DBMS vendors circumvented the OS file system altogether using raw IO while others chose to work around these restrictions. Hence all leading commercial DBMS may spread a table over multiple files or store multiple tables in a single database file. Over time, most OS file systems have evolved beyond these limitations. But the legacy influence persists and modern DBMSs still typically treat OS files as abstract storage units that map arbitrarily to database tables.

More complex is the code to handle device-specific details for maintaining temporal and spatial control as described in Section 4. A large and vibrant industry exists today based on complex storage devices that "pretend" to be disk drives, but that are in fact large hardware/software systems whose API is a legacy disk drive interface like SCSI. These systems include RAID systems and Storage Area Network (SAN) devices and tend to have very large capacities and complex performance characteristics. Administrators like these systems because they are easy to install, and often provide easily managed, bit-level reliability with fast failover. These features provide a significant sense of comfort to customers, above and beyond the promises of DBMS recovery subsystems. Large DBMS installations, for example, commonly use SANs.

Unfortunately, these systems complicate DBMS implementations. As an example, RAID systems perform very differently after a fault than they do when all disks are functioning correctly. This potentially complicates the I/O cost models for the DBMS. Some disks can operate in write-cache enabled mode, but this can lead to data corruption during hardware failure. Advanced SANs implement large battery-backed caches, in some cases nearing a terabyte, but these systems bring with them well over a million lines of microcode and considerable complexity. With complexity comes new failure modes and these problems can be incredibly difficult to detect and properly diagnose.

RAID systems also frustrate database designers by underperforming on database tasks. RAID was conceived for bytestream-oriented storage (a la UNIX files), rather than the page-oriented storage used by database systems. Hence RAID devices tend to underperform when compared with database-specific solutions for partitioning and replicating data across multiple physical devices. The *chained declustering* scheme of Gamma [43], for example, roughly coincided with the invention of RAID and performs better for a DBMS environment. Further, most databases provide DBA commands to control the partitioning of data across multiple devices, but RAID devices subvert these commands by hiding the multiple devices behind a single interface.

Many users configure their RAID devices to minimize space overheads ("RAID level 5"), when the database would perform far, far better via simpler schemes like disk mirroring ("RAID level 1"). A particularly unpleasant feature of RAID level 5 is that write performance is poor. This can cause surprising bottlenecks for users, and the DBMS vendors are often on the hook to explain or provide workarounds for these bottlenecks. For better or worse, the use (and misuse) of RAID devices is a fact that commercial DBMSs must take into account. As a result, most vendors spend significant energy tuning their DBMSs to work well on the leading RAID devices.

In the last decade, most customer deployments allocate database storage to files rather than directly to logical volumes or raw devices. But most DBMSs still support raw device access and often use this storage mapping when running high-scale transaction processing benchmarks. And, despite some of the drawback outlines above, most enterprise DBMS storage is SAN-hosted today.

7.4 Replication Services

It is often desirable to replicate databases across a network via periodic updates. This is frequently used for an extra degree of reliability: the replicated database serves as a slightly-out-of-date "warm standby" in case the main system goes down. Keeping the warm standby in a physically different location is advantageous to allow continued functioning

after a fire or other catastrophe. Replication is also often used to provide a pragmatic form of distributed database functionality for large, geographically distributed enterprises. Most such enterprises partition their databases into large geographic regions (e.g., nations or continents), and run all updates locally on the primary copies of the data. Queries are executed locally as well, but can run on a mix of fresh data from their local operations, and slightly-out-of-date data replicated from remote regions.

Ignoring hardware techniques (e.g., EMC SRDF), three typical schemes for replication are used, but only the third provides the performance and scalability needed for high-end settings. It is, of course, the most difficult to implement.

1. *Physical Replication*: The simplest scheme is to physically duplicate the entire database every replication period. This scheme does not scale up to large databases, because of the bandwidth for shipping the data, and the cost for reinstalling it at the remote site. Moreover, guaranteeing a transactionally consistent snapshot of the database is tricky. Physical replication is therefore used only as a client-side workaround at the low end. Most vendors do not encourage this scheme via any software support.
2. *Trigger-Based Replication*: In this scheme, triggers are placed on the database tables so that upon any insert, delete, or update to the table, a "difference" record is installed in a special replication table. This replication table is shipped to the remote site, and the modifications are "replayed" there. This scheme solves the problems mentioned above for physical replication, but brings a performance penalty unacceptable for some workloads.
3. *Log-Based Replication*: Log-based replication is the replication solution of choice when feasible. In log-based replication, a *log sniffer process* intercepts log writes and delivers them to the remote system. Log-based replication is implemented using two broad techniques: (1) read the log and build SQL statements to be replayed against the target system, or

(2) read log records and ship these to the target system, which is in perpetual recovery mode replaying log records as they arrive. Both mechanisms have value, so Microsoft SQL Server, DB2, and Oracle implement both. SQL Server calls the first Log Shipping and the second Database Mirroring.

This scheme overcomes all of the problems of the previous alternatives: it is low-overhead and incurs minimal or invisible performance overhead on the running system; it provides incremental updates, and hence scales gracefully with the database size and the update rate; it reuses the built-in mechanisms of the DBMS without significant additional logic; and finally, it naturally provides transactionally consistent replicas via the log's built-in logic.

Most of the major vendors provide log-based replication for their own systems. Providing log-based replication that works across vendors is much more difficult, because driving the vendors replay logic at the remote end requires an understanding of that vendor's log formats.

7.5 Administration, Monitoring, and Utilities

Every DBMS provides a set of utilities for managing their systems. These utilities are rarely benchmarked, but often dictate the manageability of the system. A technically challenging and especially important feature is to make these utilities run *online*, i.e., while user queries and transactions are in flight. This is important in the 24×7 operations that have become much more common in recent years due to the global reach of e-commerce. The traditional "reorg window" in the wee hours of the morning is typically no longer available. Hence most vendors have invested significant energy in recent years in providing online utilities. We give a flavor of these utilities here:

- *Optimizer Statistics Gathering*: Every major DBMS has some means to sweep tables and build optimizer statistics of one sort or another. Some statistics, such as histograms, are non-trivial to build in one pass without flooding memory.

For examples, see the work by Flajolet and Martin on computing the number of distinct values in a column [17].

- *Physical Reorganization and Index Construction*: Over time, access methods can become inefficient due to patterns of insertions and deletions that leave unused space. Also, users may occasionally request that tables be reorganized in the background, for example to recluster (sort) them on different columns, or to repartition them across multiple disks. Online reorganization of files and indexes can be tricky, since holding locks for any length of time must be avoided while maintaining physical consistency. In this sense it bears some analogies to the logging and locking protocols used for indexes, as described in Section 5.4. This has been the subject of several research papers [95] and patents.
- *Backup/Export*: All DBMSs support the ability to physically dump the database to backup storage. Again, since this is a long-running process, it cannot naively set locks. Instead, most systems perform some kind of "fuzzy" dump, and augment it with logging logic to ensure transactional consistency. Similar schemes can be used to export the database to an interchange format.
- *Bulk Load*: In many scenarios, massive amounts of data need to be brought quickly into the database. Rather than inserting each row one at a time, vendors supply a bulk load utility optimized for high speed data import. Typically these utilities are supported by custom code in the storage manager. For example, special bulk-loading code for B+-trees can be much faster than repeated calls to the tree-insertion code.
- *Monitoring, Tuning, and Resource Governers*: It is not unusual, even in managed environments, for a query to consume more resources than desirable. Hence most DBMSs provide tools to assist administrators in identifying and preventing these kinds of problems. It is typical to provide a SQL-based interface to DBMS performance counters via "virtual tables," which can show system status broken down by queries or by resources like locks, memory, temporary stor-

age, and the like. In some systems, it is also possible to query historical logs of such data. Many systems allow alerts to be registered when queries exceed certain performance limits, including running time, memory or lock acquisition; in some cases the triggering of an alert can cause the query to be aborted. Finally, tools like IBM's Predictive Resource Governor attempt to prevent resource-intensive queries from being run at all.

8

Conclusion

As should be clear from this paper, modern commercial database systems are grounded both in academic research and in the experiences of developing industrial-strength products for high-end customers. The task of writing and maintaining a high-performance, fully functional relational DBMS from scratch is an enormous investment in time and energy. Many of the lessons of relational DBMSs, however, translate over to new domains. Web services, network-attached storage, text and e-mail repositories, notification services, and network monitors can all benefit from DBMS research and experience. Data-intensive services are at the core of computing today, and knowledge of database system design is a skill that is broadly applicable, both inside and outside the halls of the main database shops. These new directions raise a number of research problems in database management as well, and point the way to new interactions between the database community and other areas of computing.

Acknowledgments

The authors would like to thank Rob von Behren, Eric Brewer, Paul Brown, Amol Deshpande, Cesar Galindo-Legaria, Jim Gray, Wei Hong, Matt Huras, Lubor Kollar, Ganapathy Krishnamoorthy, Bruce Lindsay, Guy Lohman, S. Muralidhar, Pat Selinger, Mehul Shah, and Matt Welsh for background information and comments on early drafts of this paper.

References

[1] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *16th International Conference on Data Engineering (ICDE)*, San Diego, CA, February 2000.

[2] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modelling: Alternatives and implications," *ACM Transactions on Database Systems (TODS)*, vol. 12, pp. 609–654, 1987.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. Frank King III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Transactions on Database Systems (TODS)*, vol. 1, pp. 97–137, 1976.

[4] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees," *Acta Informatica*, vol. 9, pp. 1–21, 1977.

[5] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A genetic algorithm for database query optimization," in *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 400–407, San Diego, CA, July 1991.

[6] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1–10, San Jose, CA, May 1995.

[7] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, 1981.

[8] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 590–594, Athens, Greece, August 1997.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[10] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of ACM Principles of Database Systems (PODS)*, 1998.

[11] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM SIGMOD Record*, March 1997.

[12] S. Chaudhuri and V. R. Narasayya, "Autoadmin 'what-if' index analysis utility," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 367–378, Seattle, WA, June 1998.

[13] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," *ACM Transactions on Database Systems (TODS)*, vol. 24, pp. 177–228, 1999.

[14] M.-S. Chen, J. Hun, and P. S. Yu, "Data mining: An overview from a database perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, 1996.

[15] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proceedings of 11th International Conference on Very Large Data Bases (VLDB)*, pp. 127–141, Stockholm, Sweden, August 1985.

[16] A. Desphande, M. Garofalakis, and R. Rastogi, "Independence is good: Dependency-based histogram synopses for high-dimensional data," in *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, February 2001.

[17] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computing System Science*, vol. 31, pp. 182–209, 1985.

[18] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten, "Fast, randomized join-order selection — why use transformations?," *VLDB*, pp. 85–95, 1994.

[19] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 9–18, San Diego, CA, June 1992.

[20] M. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes, a tutorial," in *International Conferenence on Very Large Data Bases*, 2001. www.vldb.org/conf/2001/tut4.pdf.

[21] M. N. Garofalakis and Y. E. Ioannidis, "Parallel query scheduling and optimization with time- and space-shared resources," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 296–305, Athens, Greece, August 1997.

[22] R. Goldman and J. Widom, "Wsq/dsq: A practical approach for combined querying of databases and the web," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 2000.

[23] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 102–111, Atlantic City, May 1990.

[24] G. Graefe, "Query evaluation techniques for large databases," *Computing Surveys*, vol. 25, pp. 73–170, 1993.

[25] G. Graefe, "The cascades framework for query optimization," *IEEE Data Engineering Bulletin*, vol. 18, pp. 19–29, 1995.

[26] C. Graham, "Market share: Relational database management systems by operating system, worldwide, 2005," Gartner Report No: G00141017, May 2006.

[27] J. Gray, "Greetings from a filesystem user," in *Proceedings of the FAST '05 Conference on File and Storage Technologies*, (San Francisco), December 2005.

[28] J. Gray and B. Fitzgerald, *FLASH Disk Opportunity for Server-Applications*. http://research.microsoft.com/∼Gray/papers/FlashDiskPublic.doc.

[29] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 365–394, 1976.

[30] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[32] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 47–57, Boston, June 1984.

[33] L. Haas, D. Kossmann, E. L. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," in *International Conference on Very Large Databases (VLDB)*, 1997.

[34] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, pp. 287–317, 1983.

[35] S. Harizopoulos and N. Ailamaki, "StagedDB: Designing database servers for modern hardware," *IEEE Data Engineering Bulletin*, vol. 28, pp. 11–16, June 2005.

[36] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *Proceedings of the 32nd Very Large Databases Conference (VLDB)*, 2006.

[37] J. M. Hellerstein, "Optimization techniques for queries with expensive methods," *ACM Transactions on Database Systems (TODS)*, vol. 23, pp. 113–157, 1998.

[38] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1997.

[39] J. M. Hellerstein, J. Naughton, and A. Pfeffer, "Generalized search trees for database system," in *Proceedings of Very Large Data Bases Conference (VLDB)*, 1995.

[40] J. M. Hellerstein and A. Pfeffer, "The russian-doll tree, an index structure for sets," University of Wisconsin Technical Report TR1252, 1994.

[41] C. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM (CACM)*, vol. 17, pp. 549–557, 1974.

[42] W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in xprs," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS)*, pp. 218–225, Miami Beach, FL, December 1991.

[43] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of Sixth International Conference on Data Engineering (ICDE)*, pp. 456–465, Los Angeles, CA, November 1990.

[44] Y. E. Ioannidis and Y. Cha Kang, "Randomized algorithms for optimizing large join queries," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 312–321, Atlantic City, May 1990.

[45] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 268–277, Denver, CO, May 1991.

[46] M. Kornacker, C. Mohan, and J. M. Hellerstein, "Concurrency and recovery in generalized search trees," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 62–72, Tucson, AZ, May 1997.

[47] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Tranactions on Database Systems (TODS)*, vol. 6, pp. 213–226, 1981.

[48] J. R. Larus and M. Parkes, "Using cohort scheduling to enhance server performance," in *USENIX Annual Conference*, 2002.

[49] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *ACM SIGOPS Operating Systems Review*, vol. 13, pp. 3–19, April 1979.

[50] P. L. Lehman and S. Bing Yao, "Efficient locking for concurrent operations on b-trees," *ACM Transactions on Database Systems (TODS)*, vol. 6, pp. 650–670, December 1981.

[51] A. Y. Levy, "Answering queries using views," *VLDB Journal*, vol. 10, pp. 270–294, 2001.

[52] A. Y. Levy, I. Singh Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *Proceedings of 20th International Conference on Very Large Data Bases*, pp. 96–107, Santiago, September 1994.

[53] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Sixth International Conference on Very Large Data Bases (VLDB)*, pp. 212–223, Montreal, Quebec, Canada, October 1980.

[54] G. M. Lohman, "Grammar-like functional rules for representing query optimization alternatives," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 18–27, Chicago, IL, June 1988.

[55] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier database caching for e-business," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002.

[56] S. R. Madden and M. J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *Proceedings of 12th IEEE International Conference on Data Engineering (ICDE)*, San Jose, February 2002.

[57] V. Markl, G. Lohman, and V. Raman, "Leo: An autonomic query optimizer for db2," *IBM Systems Journal*, vol. 42, pp. 98–106, 2003.

[58] C. Mohan, "Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes," in *16th International Conference on Very Large Data Bases (VLDB)*, pp. 392–405, Brisbane, Queensland, Australia, August 1990.

[59] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, pp. 94–162, 1992.

[60] C. Mohan and F. Levine, "Aries/im: An efficient and high concurrency index management method using write-ahead logging," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, (M. Stonebraker, ed.), pp. 371–380, San Diego, CA, June 1992.

[61] C. Mohan, B. G. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Transactions on Database Systems (TODS)*, vol. 11, pp. 378–396, 1986.

[62] E. Nightingale, K. Veerarghavan, P. M. Chen, and J. Flinn, "Rethink the sync," in *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[63] OLAP Market Report. Online manuscript. http://www.olapreport.com/market.htm.

[64] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 297–306, Washington, DC, May 1993.

[65] P. E. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 38–49, Tucson, May 1997.

[66] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras, "Multi-dimensional clustering: A new data layout scheme in db2," in *ACM SIGMOD International Management of Data* (San Diego, California, June 09–12, 2003) *SIGMOD '03*, pp. 637–641, New York, NY: ACM Press, 2003.

[67] D. Patterson, "Latency lags bandwidth," *CACM*, vol. 47, pp. 71–75, October 2004.

[68] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule- based query rewrite optimization in starburst," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 39–48, San Diego, June 1992.

[69] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 486–495, Athens, Greece, August 1997.

[70] M. Pöss, B. Smith, L. Kollár, and P.-Å. Larson, "Tpc-ds, taking decision support benchmarking to the next level," in *SIGMOD 2002*, pp. 582–587.

[71] V. Prabakharan, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of USENIX Annual Technical Conference*, April 2005.

[72] R. Ramakrishnan and J. Gehrke, "Database management systems," McGraw-Hill, Boston, MA, Third ed., 2003.

[73] V. Raman and G. Swart, "How to wring a table dry: Entropy compression of relations and querying of compressed relations," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2006.

[74] D. P. Reed, *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, MIT, Dept. of Electrical Engineering, 1978.

[75] A. Reiter, "A study of buffer management policies for data management systems," Technical Summary Report 1619, Mathematics Research Center, University of Wisconsin, Madison, 1976.

[76] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "System level concurrency control for distributed database systems," *ACM Transactions on Database Systems (TODS)*, vol. 3, pp. 178–198, June 1978.

[77] S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating mining with relational database systems: Alternatives and implications," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[78] R. Sears and E. Brewer, "Statis: Flexible transactional storage," in *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 22–34, Boston, June 1979.

[80] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *Proceedings of 12th IEEE International Conference on Data Engineering (ICDE)*, New Orleans, February 1996.

[81] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein, "Java support for data-intensive systems: Experiences building the telegraph dataflow system," *ACM SIGMOD Record*, vol. 30, pp. 103–114, 2001.

[82] L. D. Shapiro, "Exploiting upper and lower bounds in top-down query optimization," *International Database Engineering and Application Symposium (IDEAS)*, 2001.

[83] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill, Boston, MA, Fourth ed., 2001.

[84] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *VLDB Journal*, vol. 6, pp. 191–208, 1997.

[85] M. Stonebraker, "Retrospection on a database system," *ACM Transactions on Database Systems (TODS)*, vol. 5, pp. 225–240, 1980.

[86] M. Stonebraker, "Operating system support for database management," *Communications of the ACM (CACM)*, vol. 24, pp. 412–418, 1981.

[87] M. Stonebraker, "The case for shared nothing," *IEEE Database Engineering Bulletin*, vol. 9, pp. 4–9, 1986.

[88] M. Stonebraker, "Inclusion of new types in relational data base systems," *ICDE*, pp. 262–269, 1986.

[89] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran,

and S. Zdonik, "C-store: A column oriented dbms," in *Proceedings of the Conference on Very Large Databases (VLDB)*, 2005.

[90] M. Stonebraker and U. Cetintemel, "One size fits all: An idea whose time has come and gone," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2005.

[91] Transaction Processing Performance Council 2006. TPC Benchmark C Standard Specification Revision 5.7, http://www.tpc.org/tpcc/spec/tpcc_current.pdf, April.

[92] T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost based query scrambling for initial delays," *ACM-SIGMOD International Conference on Management of Data*, 1998.

[93] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP-19)*, Lake George, New York, October 2003.

[94] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well- conditioned, scalable internet services," in *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[95] C. Zou and B. Salzberg, "On-line reorganization of sparsely-populated b+trees," pp. 115–124, 1996.