# Anatomy of a Database System

Joseph M. Hellerstein and Michael Stonebraker

## 1 Introduction

Database Management Systems (DBMSs) are complex, mission-critical pieces of software. Today's DBMSs are based on 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 issues 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. Unfortunately, many of the architectural innovations implemented in high-end database systems are regularly reinvented both in academia and in other areas of the software industry.

There are a number of reasons why the lessons of database systems architecture are not widely known. First, the applied database systems community is fairly small. There are only a handful of commercial-grade DBMS implementations, since market forces only support a few competitors at the high end. 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 has often ignored architectural issues. The textbook presentation of database systems has traditionally focused on algorithmic and theoretical issues – which are natural to teach, study and test – without a holistic discussion of system architecture in full-fledged implementations. In sum, there is a lot of conventional wisdom about how to build database systems, but much of it has not 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. Our goal here is not to glory in the implementation details of specific components. Instead, we focus on overall system design, and stress issues not typically discussed in textbooks. For cognoscenti, this paper should be entirely familiar, perhaps even simplistic. However, our hope is that for many readers this paper will provide useful context for the algorithms and techniques in the standard literature. We assume that the reader is familiar with textbook database systems material (e.g. [53] or [61]), and with the basic facilities of modern operating systems like Solaris, Linux, or Windows.

## 1.1 Context

The most mature database systems in production are relational database management systems (RDBMSs), which serve as the backbone of infrastructure applications including banking, airline reservations, medical records, human resources, payroll, telephony, customer relationship management and supply chain management, to name a few. The advent of web-based interfaces has only increased the volume and breadth of use of relational systems, which serve as the repositories of record behind essentially all online commerce. In addition to being very important software infrastructure today, relational database systems serve as a well-understood point of reference for new extensions and revolutions in database systems that may arise in the future.

In this paper we will focus on the architectural fundamentals for supporting core relational features, and bypass discussion of the many extensions present in modern RDBMSs. Many people are unaware that commercial relational systems now encompass enormous feature sets, with support for complex data types, multiple programming languages executing both outside and inside the system, gateways to various external data sources, and so on. (The current SQL standard specification stacks up to many inches of printed paper in small type!) In the interest of keeping our discussion here manageable, we will gloss over most of these features; in particular we will not discuss system extensions for supporting complex code (stored procedures, user-defined functions, Java Virtual Machines, triggers, recursive queries, etc.) and data types (Abstract Data Types, complex objects, XML, etc.)

At heart, a typical database system has four main pieces as shown in Figure 1: a process manager that encapsulates and schedules the various tasks in the system; a statement-at-a-time query processing engine; a shared transactional storage subsystem that knits together storage, buffer management, concurrency control and recovery; and a set of shared utilities including memory management, disk space management, replication, and various batch utilities used for administration.

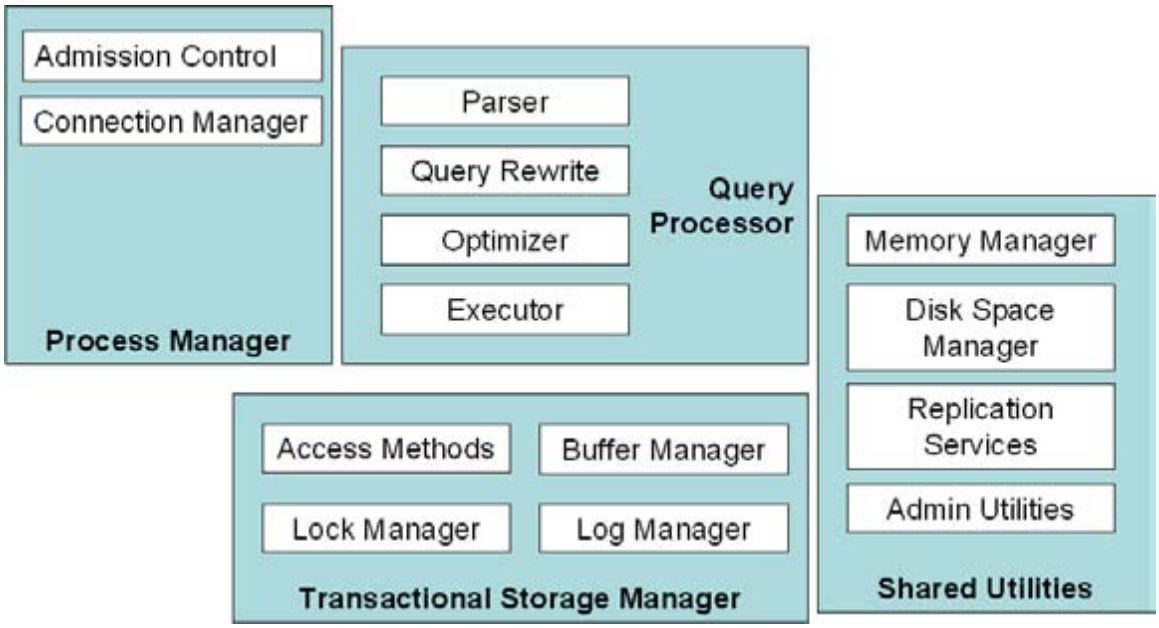


**Figure 1: Main Components of a DBMS**

### 1.2 Structure of the Paper

We begin our discussion with overall architecture of DBMS processes, including coarse structure of the software and hardware configurations of various systems, and details about the allocation of various database tasks to threads or processes provided by an operating system. We continue with the storage issues in a DBMS. In the next section we take a single query's view of the system, focusing on the query processing engine. The subsequent section covers the architecture of a transactional storage manager. Finally, we present some of the shared utilities that exist in most DBMSs, but are rarely discussed in textbooks.

## 2 Process Models and Hardware Architectures

When building any multi-user server, decisions have to be made early on regarding the organization of processes in the system. 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. We begin with a simplified framework, assuming the availability of good operating system support for lightweight threads in a uniprocessor architecture. We then expand on this simplified discussion to deal with the realities of how DBMSs implement their own threads and map them to the OS facilities, and how they manage multiprocessor configurations.

[1] Most systems are designed to be portable, but not all. Notable examples of OS-specific DBMSs are DB2 for MVS, and Microsoft SQL Server. These systems can exploit (and sometimes add!) special OS features, rather than using DBMS-level workarounds.

## 2.1 Uniprocessors and OS Threads

In this subsection we outline a somewhat simplistic approach to process models for DBMSs. Some of the leading commercial DBMSs are *not* architected this way today, but this introductory discussion will set the stage for the more complex details to follow in the remainder of Section 2.

We make two basic assumptions in this subsection, which we will relax in the subsections to come:

1. **High-performance OS threads:** We assume that the operating system provides us with a very efficient *thread* package that allows a process to have a very large number of threads. We assume that the memory overhead of each thread is small, and that context switches among threads are cheap. This is arguably true on a number of the modern operating systems, but was certainly not true when most DBMSs were first built. In subsequent sections we will describe how DBMS implementations actually work with OS threads and processes, but for now we will assume that the DBMS designers had high-performance threads available from day one.
2. **Uniprocessor Hardware:** We will assume that we are designing for a single machine with a single CPU. Given the low cost of dual-processor and four-way server PCs today, this is an unrealistic assumption even at the low end. However, it will significantly simplify our initial discussion.

In this simplified context, there are three natural process model options for a DBMS. From simplest to most sophisticated, these are:

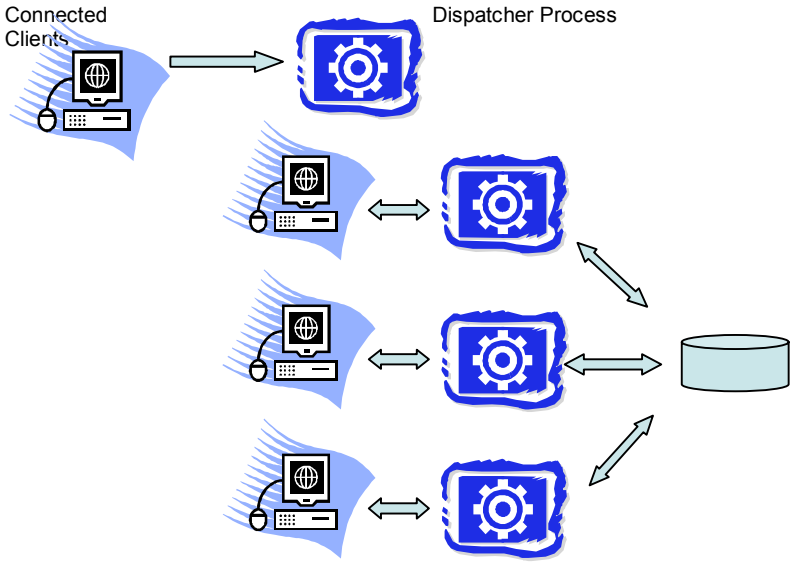


**Figure 2: Process per connection model.** Each gear icon represents a process.

1. **Process per Connection:** This was the model used in early DBMS implementations on UNIX. In this model, users run a client tool, typically on a machine across a network from the DBMS server. They use a database

connectivity protocol (e.g., ODBC or JDBC) that connects to a main dispatcher process at the database server machine, which forks a separate *process* (not a thread!) to serve that connection. This is relatively easy to implement in UNIX-like systems, because it maps DBMS units of work directly onto OS processes. The OS scheduler manages timesharing of user queries, 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. A complication of programming in this model regards the data structures that are shared across connections in a DBMS, including the lock table and buffer pool. These must be explicitly allocated in OS-supported "shared memory" accessible across processes, which requires a bit of special-case coding in the DBMS.

In terms of performance, this architecture is not attractive. It does not scale very well in terms of the number of concurrent connections, since processes are heavyweight entities with sizable memory overheads and high context-switch times. Hence this architecture is inappropriate for one of the bread-and-butter applications of commercial DBMSs: high-concurrency transaction processing. This architecture was replaced in the commercial DBMS vendors long ago, though it is still a compatibility option in many systems (and in fact the default option on installation of Oracle for UNIX).

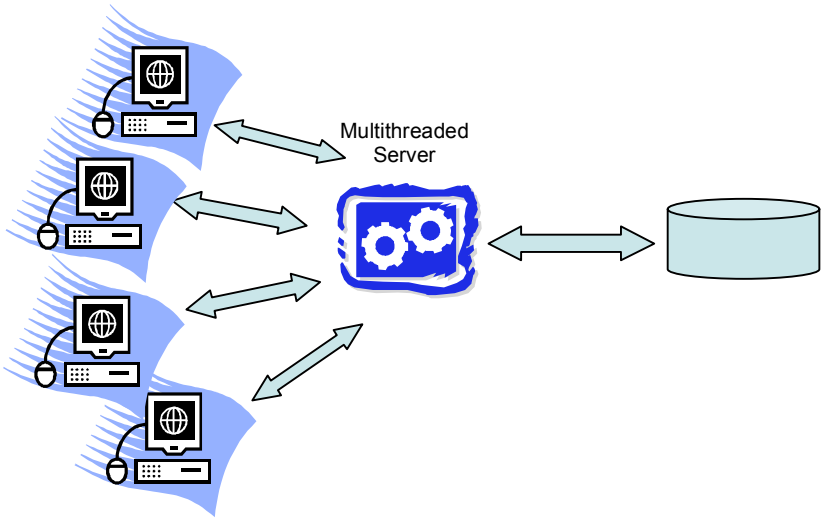


**Figure 3: Server Process model.** The multiple-gear icon represents a multithreaded process.

2. **Server Process:** This is the most natural architecture for efficiency today. In this architecture, a single multithreaded process hosts all the main activity of the DBMS. A *dispatcher* thread (or perhaps a small handful of such threads) listens for SQL commands. Typically the process keeps a pool of idle *worker threads* available, and the dispatcher assigns incoming SQL commands to idle worker threads, so that each command runs in its own thread. When a command is completed, it clears its state and returns its worker thread to the thread pool.

Shared data structures like the lock table and buffer pool simply reside in the process' heap space, where they are accessible to all threads.

The usual multithreaded 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 operating systems due to differences in threading interfaces and multi-threaded performance. Although thread API differences across operating systems have been minimized in recent years, subtle distinctions across platforms still cause hassles in debugging and tuning.

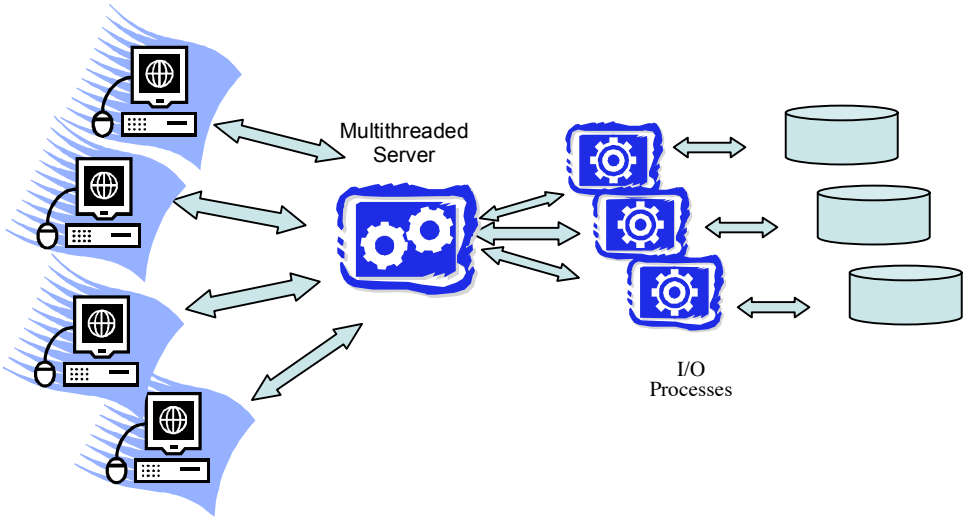


**Figure 4: Server process + I/O processes.** Note that each disk has a dedicated, single-threaded I/O process.

3. **Server Process + I/O Processes:** The Server Process model makes the important assumption that *asynchronous I/O* is provided by the operating system. This feature allows the DBMS to issue a read or write request, and work on other things while the disk device works to satisfy the request. Asynchronous I/O can also allow the DBMS to schedule an I/O reques to each of multiple disk devices; and have the devices all working in parallel; this is possible even on a uniprocessor system, since the disk devices themselves work autonomously, and in fact have their own microprocessors on board. Some time after a disk request is issued, the OS interrupts the DBMS with a notification the request has completed. Because of the separation of requests from responses, this is sometimes called a *split-phase* programming model.

   Unfortunately, asynchronous I/O support in the operating system is a fairly recent development: Linux only included asynchronous disk I/O support in the standard kernel in 2002. Without asynchronous I/O, all threads of a process must block

while waiting for any I/O request to complete, which can unacceptably limit both system throughput and per-transaction latency. To work around this issue on older OS versions, a minor modification to the Server Process model is used. Additional *I/O Processes* are introduced to provide asynchronous I/O features outside the OS. The main Server threads queue I/O requests to an I/O Process via shared memory or network sockets, and the I/O Process queues responses back to the main Server Process in a similar fashion. There is typically about one I/O Process per disk in this environment, to ensure that the system can handle multiple requests to separate devices in parallel.

### 2.1.1 Passing Data Across Threads

A good Server Process architecture provides non-blocking, asynchronous I/O. It also has dispatcher threads connecting client requests to worker threads. This design begs the question of how data is passed across these thread or process boundaries. The short answer is that various buffers are used. We describe the typical buffers here, and briefly discuss policies for managing them.

- **Disk I/O buffers:** The most common asynchronous interaction in a database is for disk I/O: a thread issues an asynchronous disk I/O request, and engages in other tasks pending a response. There are two separate I/O scenarios to consider:
    - **DB I/O requests: The Buffer Pool.** All database data is staged through the DBMS *buffer pool*, about which we will have more to say in Section 3.3**.** In Server Process architectures, this is simply a heap-resident data structure. To flush a buffer pool page to disk, a thread generates an I/O request that includes the page's current location in the buffer pool (the *frame*), and its destination address on disk. 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 frame in the buffer pool where the result can be placed. The actual reading and writing and pages into and out of frames is done asynchronously.
    - **Log I/O Requests: The Log Tail.** The database log is an array of entries stored on a set of disks. As log entries are generated during transaction processing, they are staged in a memory queue that is usually called the *log tail,* which is periodically flushed to the log disk(s) in FIFO order. In many systems, a separate thread is responsible for periodically flushing the log tail to the disk.

      The most important log flushes are those that commit transactions. A transaction cannot be reported as successfully committed until a commit log record is flushed to the log device. This means both that client code waits until the commit log record is flushed, and that DBMS server code must hold resources (e.g. locks) until that time as well. In order to amortize the costs of log writes, most systems defer them until enough are queued up, and then do a "group commit" [27] by flushing the log tail. Policies for group commit are a balance between keeping commit latency low (which favors flushing the log tail more often), and maximizing log

throughput (which favors postponing log flushes until the I/O can be amortized over many bytes of log tail).

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

  In order to support this workahead behavior, the DBMS worker thread for a query contains a pointer to a location for enqueuing results. A simple option is to assign each client to a network socket. In this case, the worker thread can use the socket as a queue for the tuples it produces. An alternative is to multiplex a network socket across multiple clients. In this case, the server process must (a) maintain its own state per client, including a communication queue for each client's SQL results, and (b) have a "coordinator agent" thread (or set of threads) available to respond to client `FETCH` requests by pulling data off of the communication queue.

## 2.2 DBMS Threads, OS Processes, and Mappings Between Them

The previous section provided a simplified description of DBMS threading models. In this section we relax the first of our assumptions above: the need for high-performance OS thread packages. We provide some historical perspective on how the problem was solved in practice, and also describe the threading in modern systems.

Most of today's DBMSs have their roots in research systems from the 1970's, and commercialization efforts from the '80's. Many of the OS features we take for granted today were unavailable to DBMS developers at the time the original database systems were built. We touched on some of these above: buffering control in the filesystem, and asynchronous I/O service. A more fundamental issue that we ignored above was the lack of high-performance threading packages. When such packages started to become available in the 1990's, they were typically OS-specific. Even the current POSIX thread standard is not entirely predictable across platforms, and recent OS research suggests that OS threads still do not scale as well as one might like ([23][37][67][68], etc.)

Hence for legacy, portability, and performance reasons, many commercial DBMSs provide their own lightweight, logical thread facility at application level (i.e. outside of the OS) for the various concurrent tasks in the DBMS. We will use the term *DBMS thread* to refer to one of these DBMS-level tasks. These 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 do all slow activities (e.g. I/Os) via non-blocking, asynchronous interfaces, and to frequently yield control to a scheduling routine (another DBMS thread) that dispatches among these tasks. This is an old idea, discussed in a retrospective sense in [38], and widely used in event-loop programming for user interfaces. It has been revisited quite a bit in the recent OS literature [23][37][67][68].

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.) [64].

Using a DBMS-level thread package raises another set of design questions. Given DBMS threads and OS process facilities (but *no* OS threads), it is not obvious how to map DBMS threads into OS processes: How many OS processes should there be? What DBMS tasks should get their own DBMS threads? How should threads be assigned to the processes? To explore this design space, we simplify things by focusing on the case where there are only two units of scheduling: DBMS threads and OS processes. We will reintroduce OS threads into the mix in Section 2.2.1.

In the absence of OS thread support, a good rule of thumb is to have one process per physical device (CPU, disk) to maximize the physical parallelism inherent in the hardware, and to ensure that the system can function efficiently in the absence of OS support for asynchronous I/O. To that end, a typical DBMS has the following set of processes:

- **One or more processes to host DBMS threads for SQL processing**. These processes host the worker DBMS threads for query processing. In some cases it is beneficial to allocate more than one such process per CPU; this is often a "tuning knob" that can be set by the database administrator.
- **One or more "dispatcher" processes.** These processes listen on a network port for new connections, and dispatch the connection requests to a DBMS thread in another process for further processing. The dispatcher also sets up session state (e.g. communication queues) for future communication on the connection. The number of dispatchers is typically another knob that can be set by the database administrator; a rule of thumb is to set the number of dispatchers to be the expected peak number of concurrent connections divided by a constant (Oracle recommends dividing by 1000.)
- **One process per database disk (I/O Process Architectures).** For platforms where the OS does not supply efficient asynchronous I/O calls, the lack of OS threads requires multiple I/O Processes, one per database disk, to service I/O requests.
- **One process per log disk (I/O Process Architectures).** For platforms with I/O Processes, there will be a process per log disk, to flush the log tail, and to read the log in support of transaction rollback.
- **One coordinator agent process per client session.** In some systems, a process is allocated for each client session, to maintain session state and handle client communication. In other systems this state is encapsulated in a data structure that is available to the DBMS threads in the SQL processes.
- **Background Utilities:** As we discuss in Section 6, DBMSs include a number of background utilities for system maintenance, including database statistics-gathering, system monitoring, log device archiving, and physical reorganization. Each of these typically runs in its own process, which is typically spawned dynamically on a schedule.

### 2.2.1 DBMS Threads, OS Threads and Current Commercial Systems

The preceding discussion assumes no support for OS threads. In fact, most modern operating systems now support reasonable threads packages. They may not provide the degree of concurrency needed by the DBMS (Linux threads were very heavyweight until recently), but they are almost certainly more efficient than using multiple processes as described above.

Since most database systems evolved along with their host operating systems, they were originally architected for single-threaded processes as we just described. As OS threads matured, a natural form of evolution was to modify the DBMS to be a single process, using an OS *thread* for each unit that was formerly an OS process. This approach continues to use the DBMS threads, but maps them into OS threads rather than OS processes. This evolution is relatively easy to code, and leverages the code investment in efficient DBMS threads, minimizing the dependency on high-end multithreading in the OS.

In fact, most of today's DBMSs are written in this manner, and can be run over either processes or threads. They abstract the choice between processes and threads in the code, mapping DBMS threads to OS-provided "dispatchable units" (to use DB2 terminology), be they processes or threads.

Current hardware provides one reason to stick with processes as the "dispatchable unit". On many architectures today, the addressable memory per process is not as large as available physical memory – for example, on Linux for x86 only 3GB of RAM is available per process. It is certainly possible to equip a modern PC with more physical memory than that, but no individual process can address all of the memory. Using multiple processes alleviates this problem in a simple fashion.

There are variations in the threading models in today's leading systems. Oracle on UNIX is configured by default to run in Process-Per-User mode, but for better performance can run in the Server Process fashion described at the beginning of Section 2.2: DBMS threads multiplexed across a set of OS processes. On Windows, Oracle uses a single OS process with multiple threads as dispatchable units: DBMS threads are multiplexed across a set of OS threads. DB2 does not provide its own DBMS threads. On UNIX platforms DB2 works in a Process-per-User mode: each user's session has its own agent process that executes the session logic. DB2 on Windows uses OS threads as the dispatchable unit, rather than multiple processes. Microsoft SQL Server only runs on Windows; it runs an OS thread per session by default, but can be configured to multiplex various "DBMS threads" across a single OS thread; in the case of SQL Server the "DBMS threads" package is actually a Windows-provided feature known as *fibers*.

## *2.3 Parallelism, Process Models, and Memory Coordination*

In this section, we relax the second assumption of Section 3.1, by focusing on platforms with multiple CPUs. Parallel hardware is a fact of life in modern server situations, and comes in a variety of configurations. We summarize the standard DBMS terminology

(introduced in [65]), and discuss the process models and memory coordination issues in each.

### 2.3.1 Shared Memory

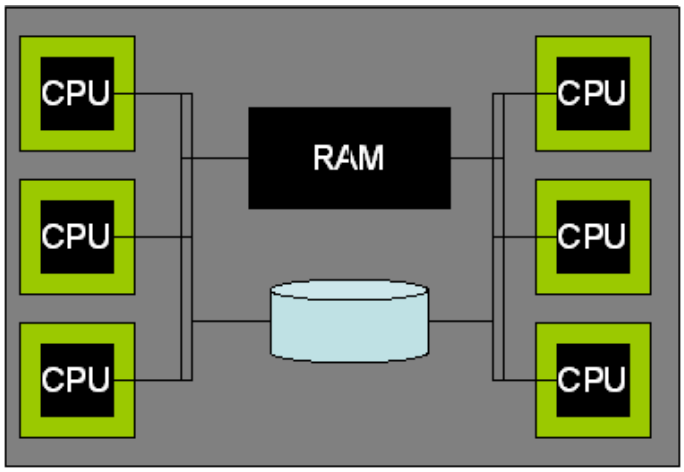


**Figure 5: Shared Memory Architecture**

A *shared-memory* parallel machine is one in which all processors can access the same RAM and disk with about 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 to hundreds of processors, but tend to be sold at an enormous premium relative to the number of compute resources provided. Massively 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 hardware is rarely the dominant factor in most companies' IT ledgers, so this cost is often deemed acceptable[2].

The process model for shared memory machines follows quite naturally from the uniprocessor Server Process approach – and 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 dispatchable units (processes or threads) across the processors, and the shared data structures continue to be accessible to all. Hence the Server Process architecture parallelizes to shared-memory machines with minimal effort. The main challenge is to modify the query execution layers described in Section 3 to take advantage of the ability to parallelize a single query across multiple CPUs.

[2] 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. Interestingly, these are typically very different career tracks, with very different training, skill sets, and responsibilities.

### 2.3.2 Shared Nothing

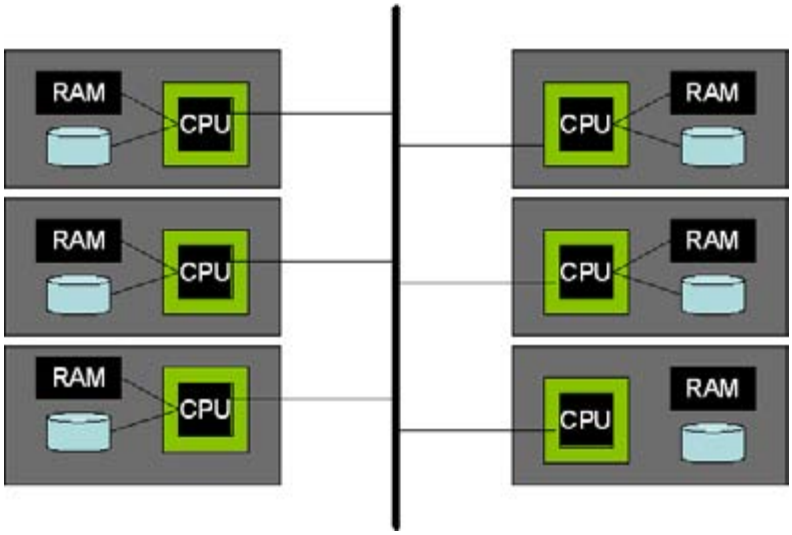


**Figure 6: Shared Nothing Architecture**

A *shared-nothing* parallel machine is made up of a cluster of single-processor machines that communicate over a high-speed network interconnect. There is no way for a given processor to directly access the memory or disk of another processor. This architecture is also fairly standard today, and has unbeatable scalability and cost characteristics. It is mostly used at the extreme high end, typically for decision-support applications on data warehouses. Shared nothing machines can be cobbled together from individual PCs, but for database server purposes they are typically sold (at a premium!) as packages including specialized network interconnects (e.g. the IBM SP2 or the NCR WorldMark machines.) In the OS community, these platforms have been dubbed "clusters", and the component PCs are sometimes called "blade servers".

Shared nothing systems provide no hardware sharing abstractions, leaving coordination of the various machines entirely in the hands of the DBMS. In these systems, each machine runs its own Server Process as above, but allows an individual query's execution to be parallelized across multiple machines. The basic architecture of these systems is to use *horizontal data partitioning* to allow each processor to execute independently of the others. For storage purposes, 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, or round-robin). Each individual machine is responsible for the access, locking and logging of the data on its local disks. During query execution, the query planner chooses how to horizontally re-partition tables across the machines to satisfy the query, assigning 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. However, good partitioning of the data is required for good performance,

which places a significant burden on the 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, there has to be explicit cross-processor coordination to handle transaction completion, to provide load balancing, and to support certain mundane maintenance tasks. For example, the processors must exchange explicit control messages for issues like distributed deadlock detection and two-phase commit [22]. 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 a hardware shutdown of the entire parallel computing machine. In a shared-nothing system, the failure of a single node will not necessarily affect other nodes, but will certainly affect the overall behavior of the DBMS, since the failed node hosts some fraction of the data in the database. There are 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 of use in scenarios where *availability* trumps *consistency*, but the best effort results generated do not have any well-defined semantics. The third approach is to employ redundancy schemes like *chained declustering* [32], which spread copies of tuples across multiple nodes in the cluster. These techniques are designed to tolerate a number of failures without losing data. In practice, however, these techniques are not provided; commercial vendors offer coarser-grained redundancy solutions like database replication (Section 6.3), which maintain a copy of the entire database in a separate "standby" system.

### 2.3.3 Shared Disk

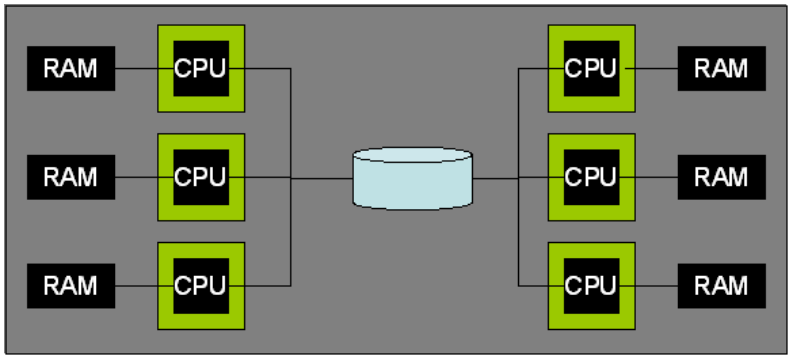


**Figure 7: Shared Disk Architecture**

A *shared-disk* parallel machine is one in which all processors can access the same disks with about the same performance, but are unable to access each other's RAM. This architecture is quite common in the very largest "single-box" (non-cluster) multiprocessors, and hence is important for very large installations – especially for

Oracle, which does not sell a shared-nothing software platform. Shared disk has become an increasingly attractive approach in recent years, with the advent of Network Attached Storage devices (NAS), which allow a storage device on a network to be mounted by a set of nodes.

One key advantage of shared-disk systems over shared-nothing is in usability, since DBAs of shared-disk systems do not have to consider partitioning tables across machines. Another feature of a shared-disk architecture is that the failure of a single DBMS processing node does not affect the other nodes' ability to access the full database. This is in contrast to both shared-memory systems that fail as a unit, and shared-nothing systems that lose at least some data upon a node failure. Of course this discussion puts more emphasis on the reliability of the storage nodes.

Because there is no partitioning of the data in a shared disk system, data can be copied into RAM and modified on multiple machines. Unlike shared-memory systems there is no natural place to coordinate this sharing of the data – each machine has its own local memory for locks and buffer pool pages. Hence there is a need to explicitly coordinate data sharing across the machines. Shared-disk systems come with a distributed lock manager facility, and a *cache-coherency* protocol for managing the distributed buffer pools [7]. These are complex pieces of code, and can be bottlenecks for workloads with significant contention.

### 2.3.4 NUMA

*Non-Uniform Memory Access (NUMA)* architectures are somewhat unusual, but available from vendors like IBM. They provide a shared memory system where the time required to access some remote memory can be much higher than the time to access local memory. Although NUMA architectures are not especially popular today, they do bear a resemblance to shared-nothing clusters in which the basic building block is a small (e.g. 4-way) multiprocessor. Because of the non-uniformity in memory access, DBMS software tends to ignore the shared memory features of such systems, and treats them as if they were (expensive) shared-nothing systems.

## *2.4 Admission Control*

We close this section with one remaining issue related to supporting multiple concurrent requests. As the workload is increased in any multi-user system, performance will increase up to some maximum, and then begin to decrease radically as the system starts to "thrash". As in operating system settings, 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, which like to use 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 restarted [2]. Hence any good multi-user system has an *admission* control policy, which does not admit

new clients unless the workload will stay safely below the maximum that can be handled without thrashing. With a good admission controller, a system will display *graceful degradation* under overload: transactions latencies will increase proportionally to their arrival rate, but throughput will remain at peak.

Admission control for a DBMS can be done in two tiers. First, there may be a simple admission control policy 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, and minimizes unnecessary invocations of the query parser and optimizer. In some DBMSs this control is not provided, under the assumption that it is handled by some other piece of software interposed between clients and the DBMS: e.g. an application server, transaction processing monitor, or web server.

The second layer of admission control must be implemented directly within the core DBMS query processor. This *execution* admission controller runs after the query is parsed and optimized, and determines whether a query is postponed or begins execution. The execution admission controller is aided by information provided by the query optimizer, which can estimate the resources that a query will require. In particular, the optimizer's query plan can specify (a) the disk devices that the query will access, and an estimate of the number of random and sequential I/Os per device (b) 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 (c) estimates about the memory footprint of the query data structures, including space for sorting and hashing tables. As noted above, this last metric is often the key for an admission controller, since memory pressure is often the main cause of thrashing. Hence many DBMSs use memory footprint as the main criterion for admission control.

## 2.5 Standard Practice

As should be clear, there are many design choices for process models in a DBMS, or any large-scale server system. However due both to historical legacy and the need for extreme high performance, a few standard designs have emerged

To summarize the state of the art for uniprocessor process models:

- Modern DBMSs are built using both "Process-per-User" and "Server Process" models; the latter is more complex to implement but allows for higher performance in some cases.
- Some Server Process systems (e.g. Oracle and Informix) implement a DBMS thread package, which serves the role taken by OS threads in the model of Section 3.1. When this is done, DBMS threads are mapped to a smaller set of "dispatchable units" as described in Section 3.2.
- Dispatchable units can be different across OS platforms as described in Section 3.2.1: either processes, or threads within a single process.

In terms of parallel architectures, today's marketplace supports a mix of Shared-Nothing, Shared-Memory and Shared-Disk architectures. As a rule, Shared-Nothing architectures excel on price-performance for running complex queries on very large databases, and

hence they occupy a high-end niche in corporate decision support systems. The other two typically perform better at the high end for processing multiple small transactions. The evolution from a uniprocessor DBMS implementation to a Shared-Nothing implementation is quite difficult, and at most companies was done by spawning a new product line that was only later merged back into the core product. Oracle still does not ship a Shared-Nothing implementation.

# 3 Storage Models

In addition to the process model, another basic consideration when designing a DBMS is the choice of the persistent storage interface to use. There are basically two options: the DBMS can interact directly with the device drivers for the disks, or the DBMS can use the typical OS file system facilities. This decision has impacts on 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.

## *3.1 Spatial Control*

Sequential access to disk blocks is between 10 and 100 times faster than random access. This gap is increasing quickly. Disk density – and hence sequential bandwidth – improves following Moore's Law, doubling every 18 months. Disk arm movement is improving at a much slower rate. As a result, it is critical for the DBMS storage manager to place blocks on the disk so that important queries can access data sequentially. Since the DBMS can understand its workload 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 issue low-level storage requests directly to the "raw" disk device interface, since disk device addresses typically correspond closely to physical proximity of storage locations. Most commercial database systems offer this functionality for peak performance. Although quite effective, this technique has some drawbacks. First, it requires the DBA to devote entire disks to the DBMS; this used to be frustrating when disks were very expensive, but it has become far less of a concern today. Second, "raw disk" access interfaces are often OS-specific, which can make the DBMS more difficult to port. However, this is a hurdle that most commercial DBMS vendors chose to overcome years ago. Finally, developments in the storage industry like RAID, Storage Area Networks (SAN), and Network-Attached Storage (NAS) have become popular, to the point where "virtual" disk devices are the norm in many scenarios today – the "raw" device interface is actually being intercepted by appliances or software that reposition data aggressively on 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 6.2.

An alternative to raw disk access is for the DBMS to create a very large file in the OS file system, and then manage positioning of data in the offsets of that file. This offers reasonably good performance. In most popular filesystems, if you allocate a very large file on an empty disk, the offsets in that file will correspond fairly well to physical

proximity of storage regions. Hence this is a good approximation to raw disk access, without the need to go directly to the 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. However, using the filesystem interface has other ramifications, which we discuss in the next subsection.

It is worth noting that in either of these schemes, the size of a database page is a tunable parameter that can be set at the time of database generation; it should be a multiple of the sized offered by typical disk devices. If the filesystem is being used, special interfaces may be required to write pages of a different size than the filesystem default; the POSIX *mmap/msync* calls provide this facility. A discussion of the appropriate choice of page sizes is given in the paper on the "5-minute rule" [20].

## 3.2 Temporal Control: Buffering

In addition to controlling *where* on the disk data should lie, 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: the DBMS cannot ensure correct transactional semantics without explicitly controlling the timing of disk writes. As we will discuss in Section 5, 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* (postponed, batched writes), and these are often poorly-suited to DBMS access patterns. File system logic depends on the contiguity of *physical* byte offsets in files to make decisions about reads and writes. DBMS-level I/O facilities can support *logical* decisions based on the DBMS' behavior. For example, the stream of reads in a query is often predictable to the DBMS, but not physically contiguous on the disk, and hence not visible via the OS read/write API. Logical DBMS-level read-ahead can occur when scanning the leaves of a B+-tree, for example. Logical read-aheads are easily achieved in DBMS logic by a query thread issuing I/Os in advance of its needs – the query 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 buffer (often called the log "tail"), based on considerations that mix issues like lock contention with I/O throughput. This mix of information is available to the DBMS, but not to the OS file system.

The final performance issues are "double buffering" and the extreme 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, effectively limiting the memory available for doing useful work. Second, it wastes time, 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, about which we will say more shortly. On writes, both of these copies are required in reverse. Copying data in memory can be a serious bottleneck in DBMS software today. This fact is often a surprise to database students, who assume that main-memory operations are "free" compared to disk I/O. But in practice, *a well-tuned database installation is typically not I/O-bound.* This is achieved in high-end installations by purchasing the right mix of 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 appetite of all the processors in the system. Once this kind of "system balance" is achieved, I/O latencies cease to be a 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 (which follows Moore's law) and RAM access speed (which trails Moore's law significantly).

The problems of OS buffering have been well-known in the database research literature [64] and the industry for some time. Most modern operating systems now provide hooks (e.g. the POSIX *mmap/msync/madvise* calls) for programs like database servers to circumvent double-buffering the file cache, ensuring that writes go through to disk when requested, that double buffering is avoided, and that some alternate replacement strategies can be hinted at by the DBMS.

## 3.3 Buffer Management

In order to provide efficient access to database pages, every DBMS implements a large shared buffer pool in its own memory space. The buffer pool is organized as an array of *frames*, each frame being a region of memory the size of a database disk block. Blocks are copied in native format from disk directly into frames, manipulated in memory in native format, and written back. This translation-free approach avoids CPU bottlenecks in "marshalling" and "unmarshalling" data to/from disk; perhaps more importantly, the fixed-sized frames sidestep complexities of external memory fragmentation and compaction that are associated with generic memory management.

Associated with the array of frames is an array of metadata called a *page table*, with one entry for each frame. The page table contains the disk location for the page currently in each frame, 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* used for choosing pages to evict on overflow. It also contains a *pin count* for the page in the frame; the page is not candidate for page replacement unless the pin count is 0. This allows tasks to

(hopefully briefly) "pin" pages into the buffer pool by incrementing the pin count before manipulating the page, and decrementing it thereafter.

Much research in the early days of relational systems focused on the design of page replacement policies. The basic tension surrounded the looping access patterns resulting from nested-loops joins, which scanned and rescanned a heap file larger than the buffer pool. For such looping patterns, recency of reference is a pessimal predictor of future reuse, so OS page replacement schemes like LRU and CLOCK were well known to perform poorly for database queries [64]. A variety of alternative schemes were proposed, including some that attempted to tune the replacement strategy via query plan information [10]. Today, most systems use simple enhancements to LRU schemes to account for the case of nested loops; one that appears in the research literature and has been implemented in commercial systems is LRU-2 [48]. Another scheme used in commercial systems is to have a 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 [55][10].

### 3.4 Standard Practice

In the last few years, commercial filesystems have evolved to the point where they can now support database storage quite well. The standard usage model is to allocate a single large file in the filesystem on each disk, and let the DBMS manage placement of data within that file via interfaces like the *mmap* suite. In this configuration, modern filesystems now offer reasonable spatial and temporal control to the DBMS. This storage model is available in essentially all database system implementations. However, the raw disk code in many of the DBMS products long predates the maturation of filesystems, and provides explicit performance control to the DBMS without any worry about subtle filesystem interactions. Hence raw disk support remains a common high-performance option in most database systems.

## 4 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. We start with the query processor.

A relational query engine takes a declarative SQL statement, validates it, optimizes it into a procedural dataflow implementation plan, and (subject to admission control) executes that dataflow on behalf of a client program, which fetches ("pulls") the result tuples, typically one at a time or in small batches. The components of a relational query engine are shown in Figure 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 described in Section 5. The only exception to this rule is that the query processor must explicitly pin and unpin buffer pool pages when manipulating them, as we note below. In this section

we focus on the common case SQL commands: "DML" statements including SELECT, INSERT, UPDATE and DELETE.

## 4.1 Parsing and Authorization

Given an SQL statement, the main tasks for the parser are to check that the query is correctly specified, to convert it into an internal format, and to check that the user is authorized to execute the query. Syntax checking is done naturally as part of the parsing process, during which time the parser generates an internal representation for the query.

The parser handles queries one "SELECT" block at a time. First, it considers each of the table references in the FROM clause. It canonicalizes each table name into a *schema.tablename* format; users have default schemas which are often omitted from the query specification. It then invokes the *catalog manager* to check that the table is registered in the system catalog; while so checking it may also cache metadata about the table in internal query data structures. Based on information about the table, it then uses the catalog to check that attribute references are correct. The data types of attributes are used to drive the (rather intricate) disambiguation logic for overloaded functional expressions, comparison operators, and constant expressions. For example, in 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, which may be an integer, a floating-point number, or a "money" value. Additional standard SQL syntax checks are also applied, including the 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 correctly, the next phase is to check authorization. Again, the catalog manager is invoked to ensure that the user has the appropriate permissions (SELECT/DELETE/INSERT/UPDATE) on the tables in the query. Additionally, integrity constraints are consulted to ensure that any constant expressions in the query do not result in constraint violations. For example, an UPDATE command may have a clause of the form "SET EMP.salary = -1". If there is an integrity constraint specifying positive values for salaries, the query will not be authorized for execution.

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

### 4.1.1 A Note on Catalog Management

The database catalog is a form of *metadata:* information about the data in the system. The catalog is itself stored as a set of tables in the database, recording the names of basic entities in the system (users, schemas, tables, columns, indexes, etc.) and their relationships. 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 few years!)

For efficiency, basic catalog data is treated somewhat differently from normal tables. High-traffic portions of the catalog are often materialized in main memory at bootstrap time, 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.

It is worth noting that catalogs can become formidably large in commercial applications. One major Enterprise Resource Planning application generates over 30,000 tables, with between 4 and 8 columns per table, and typically two or three indexes per table.

## 4.2 Query Rewrite

The query rewrite module is responsible for a number of tasks related to simplifying and optimizing the query, typically without changing its semantics. The key in all these tasks is that they can be carried out without accessing the data in the tables – all of these techniques rely only on the query and on metadata in the catalog. Although we speak of "rewriting" the query, in fact most rewrite systems operate on internal representations of the query, rather than on the actual text of a SQL statement.

- **View rewriting:** The most significant role in rewriting is to handle views. The rewriter takes each view reference that appeared in the FROM clause, and gets the view definition from the catalog manager. It then rewrites the query to remove the view, replacing it with the tables and predicates referenced by the view, and rewriting any predicates that reference the view to instead reference columns from the tables in the view. This process is applied recursively until the query is expressed exclusively over base tables. This view expansion technique, first proposed for the set-based QUEL language in INGRES [63], requires some care in SQL to correctly handle duplicate elimination, nested queries, NULLs, and other tricky details [51].
- **Constant arithmetic evaluation:** Query rewrite can simplify any arithmetic expressions that do not contain tuple variables: e.g. "R.x < 10+2" is rewritten as "R.x < 12".
- **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: for example, a predicate like "NOT Emp.Salary >

1000000" may be rewritten as "Emp.Salary <= 1000000". These logical rewrites can even short-circuit query execution, via simple satisfiability tests: for example, the expression "Emp.salary < 75000 AND Emp.salary > 1000000" can be replaced with FALSE, possibly allowing the system to return an empty query result without any accesses to 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 – e.g. the query above may have resulted from a query for low-paid employees over a view called "Executives".

An additional, important logical rewrite uses the transitivity of predicates to induce new predicates: e.g. "R.x < 10 AND R.x = S.y" 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 there are foreign key constraints from 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. 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, and hence the join. Again, such seemingly implausible scenarios often arise naturally via views – for example, a user may submit a query about employee attributes over a view EMPDEPT that joins those two tables. Semantic optimizations can also lead to short-circuited query execution, when constraints on the tables are incompatible with query predicates.
- **Subquery flattening and other heuristic rewrites:** In many systems, queries are rewritten to get them into a form that the optimizer is better equipped to handle. In particular, most optimizers operate on individual SELECT-FROM-WHERE query blocks in isolation, forgoing possible opportunities to optimize across blocks. Rather than further complicate query optimizers (which are already quite complex in commercial DBMSs), a natural heuristic is to flatten nested queries when possible to expose further opportunities for single-block optimization. This turns out to be very tricky in some cases in SQL, due to issues like duplicate semantics, subqueries, NULLs and correlation [51][58]. Other heuristic rewrites are possible across query blocks as well – for example, predicate transitivity can allow predicates to be copied across subqueries [40]. It is worth noting that the flattening of correlated subqueries is especially important for achieving good performance in parallel architectures, since the "nested-loop" execution of correlated subqueries is inherently serialized by the iteration through the loop.

When complete, the query rewrite module produces an internal representation of the query in the same internal format that it accepted at its input.

## 4.3 Optimizer

Given an internal representation of a query, the job of the query optimizer is to produce an efficient *query plan* for executing the query (Figure 8). A query plan can be thought of as a dataflow diagram starting from base relations, piping data through a graph of query *operators*. In most systems, queries are broken into SELECT-FROM-WHERE query blocks. The optimization of each individual query block is done using techniques similar to those described in the famous paper by Selinger, et al. on the System R optimizer [57]. Typically, at the top of each query block a few operators may be added as post-processing to compute GROUP BY, ORDER BY, HAVING and DISTINCT clauses if they exist. Then the various blocks are stitched together in a straightforward fashion.

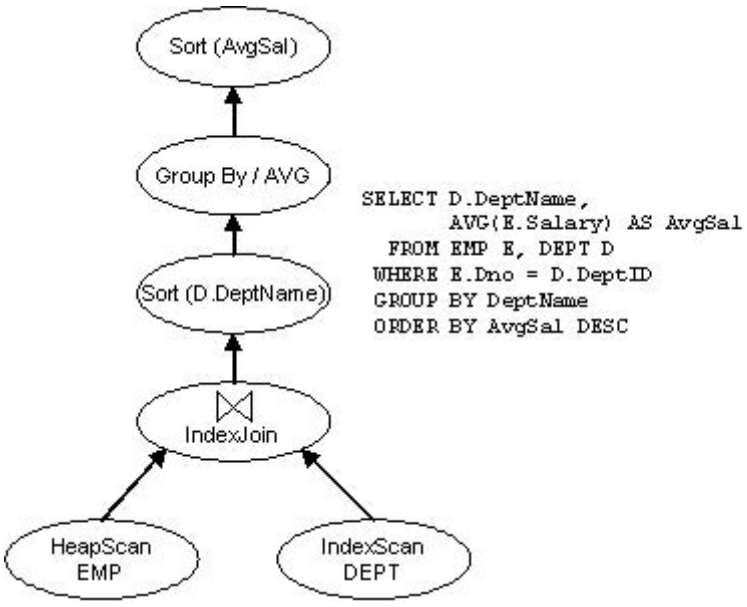


**Figure 8: A Query Plan.** Note that only the main physical operators are shown.

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 [63], but Moore's law and software engineering have vindicated the INGRES decision to some degree. In order to enable cross-platform portability, every system now compiles queries into some kind of interpretable data structure; the only difference across systems these days is the level of abstraction. In some systems the query plan is a very lightweight object, not unlike a relational algebra expression 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 algebra expressions. For simplicity in our discussion, we will 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, and all systems extend it in a number of dimensions. We consider some of the main extensions here.

- **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, and hence both options are considered in most systems.
- **Selectivity estimation:** The selectivity estimation techniques in the Selinger paper are naïve, based on simple table and index cardinalities. Most systems today have a background process that periodically analyzes and summarizes the distributions of values in attributes via histograms and other summary statistics. 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 have been proposed in the literature in recent years to incorporate issues like dependencies among columns [52] [11]; these innovations have yet to show up in products. One reason for the slow adoption of these schemes is a flaw in the industry benchmarks: the data generators in benchmarks like TPC-H generate independent values in columns, and hence do not encourage the adoption of technology to handle "real" data distributions. Nonetheless, the benefits of improved selectivity estimation are widely recognized: as noted by Ioannidis and Christodoulakis, errors in selectivity early in optimization propagate multiplicatively up the plan tree, resulting in terrible subsequent estimations [32]. Hence improvements in selectivity estimation often merit the modest implementation cost of smarter summary statistics, and a number of companies appear to be moving toward modeling dependencies across columns.
- **Search Algorithms:** Some commercial systems – notably those of Microsoft and Tandem – discard Selinger's dynamic programming algorithm in favor of a goal-directed "top-down" search scheme based on the Cascades framework [17]. Top-down search can in some instances lower the number of plans considered by an optimizer [60], 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 exponential in the number of tables in a query.

  It is also important to note that some systems fall back on heuristic search schemes for queries with "too many" tables. Although there is an interesting research literature of randomized query optimization heuristics [34][5][62], the heuristics used in commercial systems tend to be proprietary, and (if rumors are to be believed) 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) is entirely heuristic and relies 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 commercial DBMS today has some support for parallel processing, and most support "intra-query" parallelism: the ability to speed up a single query via 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) multiple separate computers on a high-speed network. The standard approach was proposed by Hong and Stonebraker [31]and uses two phases: first a traditional single-site optimizer is invoked to pick the best single-site plan, and then this plan is scheduled across the multiple processors. Research has been published on this latter phase [14][15] though it is not clear to what extent these results inform standard practice – currently this seems to be more like art than science.
- **Extensibility:** Modern SQL standards include user-defined types and functions, complex objects (nested tuples, sets, arrays and XML trees), and other features. Commercial optimizers try to handle these extensions with varying degrees of intelligence. One well-scoped issue in this area is to incorporate the costs of expensive functions into the optimization problem as suggested in [29]. In most commercial implementations, simple heuristics are still used, though more thorough techniques are presented in the research literature [28][9]. Support for complex objects is gaining importance as nested XML data is increasingly stored in relational engines. This has generated large volumes of work in the object-oriented [50] and XML [25] query processing literature.

  Having an extensible version of a Selinger optimizer as described by Lohman [42] can be useful for elegantly introducing new operators into the query engine; this is presumably the approach taken in IBM's products. A related approach for top-down optimizers was developed by Graefe [18][17], and is likely used in Microsoft SQL Server.
- **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 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 [8].

### 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, and store the resulting plan in a catalog table. This is even possible for embedded 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 have some "typical" values. Query preparation is especially useful for form-driven, canned queries: 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. In practice, this feature is used far more heavily than ad-hoc queries that are optimized at runtime.

As a database evolves, 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 catalog of stored plans, 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*. As a result, they will not reoptimize 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 reoptimize plans quite aggressively: they may even reoptimize, for example, if the value distribution of a column changes significantly, since this may affect the selectivity estimates, and hence the choice of the best plan. 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, and is in some sense self-reinforcing. 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 its mind 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. But the original philosophies tend to peek out in the system architecture, and in the way that the architecture affects the use of the systems by DBAs and database programmers.

## 4.4 Executor

A query executor is given a fully-specified query plan, which is a fixed, directed dataflow graph connecting operators that encapsulate base-table access and various query execution algorithms. In some systems this dataflow graph is already compiled into op-codes by the optimizer, in which case the query executor is basically a runtime interpreter. In other systems a representation of the dataflow graph is passed to the query executor, which recursively invokes procedures for the operators based on the graph layout. We will focus on this latter case; the op-code approach essentially compiles the logic we described here into a program.

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

**Figure 9: Iterator superclass pseudocode.**

Essentially all modern query executors employ the *iterator* model, which was used in the earliest relational systems. Iterators are most simply described in an object-oriented fashion. All operators in a query plan – the nodes in the dataflow graph – are implemented as objects from the superclass `iterator`. A simplified definition for an iterator is given in Figure 9. Each iterator specifies its inputs, which define the edges in the dataflow graph. Each query execution operator is implemented as a subclass of the iterator class: the set of subclasses in a typical system might include filescan, indexscan, nested-loops join, sort, 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 there is no need to write special-case code for particular combinations of iterators.

Graefe provides more details on iterators in his query processing survey [18]. The interested reader is encouraged to examine the open-source PostgreSQL code base, which includes 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, returning a tuple reference to the callee 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 there is no need for queues or rate-matching between iterators. This makes relational query executors clean to implement and easy to debug, and is a contrast with dataflow architectures in other environments, e.g. networks, which rely on various protocols for queueing and feedback between concurrent producers and consumers.

The single-threaded iterator architecture is also quite efficient for single-site query execution. In most database applications, the performance metric of merit is time to query completion. 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.[3]

[3] This assumes that iterators never block waiting for I/O requests. As noted above, I/O prefetching is typically handled by a separate thread. In the cases where prefetching is ineffective, there can indeed be inefficiencies in the iterator model. This is typically not a

As we mentioned previously, support for parallel query execution is standard in most modern DBMSs. Fortunately, this support can be provided with essentially no changes to the iterator model or a query execution architecture, by encapsulating parallelism and network communication within special *exchange* iterators, as described by Graefe [16].

### 4.4.2 Where's the Data?

Our discussion of iterators has conveniently sidestepped any questions of memory allocation for in-flight data; we never specified how tuples were stored in memory, or how they were passed from iterator to iterator. In practice, each iterator has a fixed number of *tuple descriptors* pre-allocated: 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 dynamically allocates memory, which raises the question of where the actual tuples being referenced are stored in memory.

There are two alternative answers to this question. The first possibility is that base-table tuples can reside in pages in the buffer pool; we will call these BP-tuples. If an iterator constructs a tuple descriptor referencing a BP-tuple, it must increment the pin count of the tuple's 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 will call this an M-tuple, It may construct an M-tuple by copying columns from the buffer pool (the copy bracketed by a pin/unpin pair), and/or by evaluating expressions (e.g. arithmetic expressions like "EMP.sal * 0.1") in the query specification.

An attractive design pitfall 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, which simplifies the executor code. It 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, exclusive use of M-tuples can be a major performance problem, since memory copies are often a serious bottleneck in high-performance systems, as noted in Section 3.2.

On the other hand, there are cases where constructing an M-tuple makes sense. It is sometimes beneficial to copy a tuple out of the buffer pool if it will be referenced for a long period of time. 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.

---

big problem in single-site databases, though it arises frequently when executing queries over remote tables [16][43].

The upshot of this discussion is that it is most efficient 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 – i.e., read-only SQL statements. Another class of DML statements modify data: INSERT, DELETE and UPDATE statements. Typically, execution plans for these statements 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 are complicated by the fact that they both query and modify the same data. This mix of reading and writing the same table (possibly multiple times) raises some complications. A simple example is the notorious "Halloween problem", 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 indexscan iterator over the Emp.salary field into an update iterator (the left-hand side of Figure 10); the pipelining provides good I/O locality, because it modifies tuples just after they are fetched from the B+-tree. However, this pipelining can also result in the indexscan "rediscovering" a previously-modified tuple that moved rightward in the tree after modification – resulting in 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.

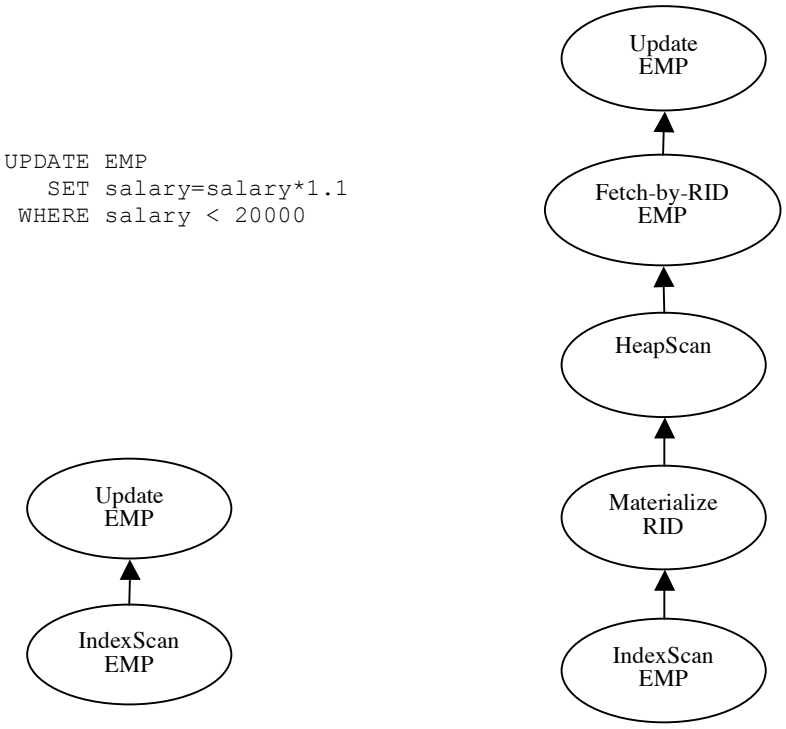


**Figure 10: Two query plans for updating a table via an IndexScan.** The plan on the left is susceptible to the Halloween problem. The plan on the right is safe, since it identifies all tuples to be updated before doing any updates.

SQL semantics forbid this behavior: an 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, but this can be quite inefficient in some cases. Another technique is to use a batch read-then-write scheme, which interposes Record-ID materialization and fetching operators between the index scan and the data modification operators in the dataflow (right-hand side of Figure 10.) This 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, feeding the resulting tuple to the data modification operator. In most cases if an index was chosen by the optimizer, it implies that only a few tuples are being changed, and 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) multiversion support from the storage engine.[54]

## *4.5 Access Methods*

The access methods are the routines for managing access to the various disk-based data structures supported by the system, which typically included unordered files ("heaps") of tuples, and various kinds of indexes. All commercial database systems include B+-tree

indexes and heap files. Most systems are beginning to introduce some rudimentary support for multidimensional indexes like R-trees [24]. Systems targeted at read-mostly data warehousing workloads usually include specialized bitmap variants of indexes as well [49].

The basic API provided by an access method is an iterator API, with the `init()` routine expanded to take a "search predicate" (or in the terminology of System R, a "search argument", or 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 there are no more tuples satisfying 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 try the next tuple on the page by reinvoking `get_next()`. This logic involves a number of CPU cycles simply doing 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. 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.

A special SARG is available in all access methods to `FETCH` a tuple directly by its physical *Record ID (RID)*. `FETCH`-by-RID is required to support secondary indexes and other schemes that "point" to tuples, and subsequently need to dereference those pointers.

In contrast to all other iterators, access methods have deep interactions with the concurrency and recovery logic surrounding transactions. We discuss these issues next.

# 5 Transactions: Concurrency Control and Recovery

Database systems are often accused of being enormous, monolithic pieces of software 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 narrow interfaces in between. This is particularly true of the various components of query processing described in the previous section. The parser, rewrite engine, optimizer, executor and access methods all represent fairly independent pieces of code with well-defined, narrow interfaces that are "published" internally between development groups.

The truly monolithic piece of a DBMS is the transactional storage manager, which 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 knowledgable about these systems should read – at a minimum – a basic undergraduate database textbook [53], the journal article on the ARIES log protocol [45], and one serious article on transactional index concurrency *and* logging [46] [35]. More advanced readers will want to leaf through the Gray and Reuter textbook on transactions [22]. To really become an expert, this reading has to be followed by an implementation effort! We will not focus on algorithms here, but rather overview the roles of these various components, focusing on the system infrastructure that is often ignored in the textbooks, and highlighting the *inter-dependencies* between the components.

## 5.1 A Note on ACID

Many people are familiar with the term "ACID transactions", a mnemonic due to Härder and Reuter [26]. ACID stands for *Atomicity, Consistency, Isolation,* and *Durability*. These terms were not formally defined, and theory-oriented students sometimes spend a great deal of time trying to tease out exactly what each letter means. The truth is that these are not mathematical axioms that combine to guarantee transactional consistency, so carefully distinguishing the terms may not be a worthwhile exercise. Despite the informal nature, the ACID acronym is useful to organize a discussion of transaction systems.

- Atomicity is the "all or nothing" guarantee for transactions – either all of a transaction's actions are visible to another transaction, or none are.
- Consistency is an application-specific guarantee, which is typically captured in a DBMS by SQL integrity constraints. 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 updates. As a result, applications need not be coded "defensively" to worry about the "dirty data" of other concurrent transactions.
- Durability is a guarantee that the updates of a committed transaction will be visible in the database to subsequent transactions, until such time as they are overwritten by another committed transaction.

Roughly speaking, modern DBMSs implement Isolation via *locking* and Durability via *logging*; Atomicity is 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.

## 5.2 Lock Manager and Latches

*Serializability* is the well-defined textbook notion of correctness for concurrent transactions: a sequence of interleaved actions for multiple committing transactions must correspond to some serial execution of the transactions. Every commercial relational DBMS implements serializability via strict two-phase locking (2PL): transactions acquire locks on objects before reading or writing them, and release all locks at the time of transactional commit or abort. The lock manager is the code module responsible for providing the facilities for 2PL. As an auxiliary to database locks, lighter-weight *latches* are also provided for mutual exclusion.

We begin our discussion with locks. 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 are managed by the DBMS. 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. 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 [21].

The lock manager supports two basic calls; `lock(lockname, transactionID, mode)`, and `remove_transaction(transactionID)`. Note that because of the strict 2PL protocol, there need 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 consistency than serializability, 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 [46].

To support these calls, the lock manager maintains two data structures. A global *lock table* is maintained to hold locknames 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 `current_mode` flag to indicate the lock mode, and a `waitqueue` of lock request pairs `(transactionID, mode)`. In addition, it maintains a *transaction table* keyed by transactionID, which contains two items for each transaction *T*: (a) 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 (b) 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 look for waits-for cycles. Upon detection of a deadlock, the deadlock detector aborts one of the deadlocked transaction (the decision of which deadlocked transaction to abort is based on heuristics that have been studied in the research literature [55].) In shared-nothing and shared-disk systems, distributed deadlock detection facilities are required as well [47]. A more description of a lock manager implementation is given in Gray and Reuter's text [22].

In addition to two-phase locks, every DBMS also supports a lighter-weight mutual exclusion mechanism, typically called a *latch*. Latches are more akin to monitors [30] 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 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.
- 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.
- Latch calls take a few dozen CPU cycles, lock requests take hundreds of CPU cycles.

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 `current_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 [46].

### 5.2.1 Isolation Levels

Very early in the development of the transaction concept, there were attempts to provide more 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" [21]. 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[4].
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: Fully 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, 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, but sees new "phantom" tuples on re-access that were not seen on the first access.[5] 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 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 in 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 [36] or multi-version [54] 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 various 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),

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

[5] Despite the spooky similarity in names, the phantom problem has nothing to do with the Halloween problem of Section 4.4.

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. 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 there is 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 (though these schemes typically coexist in systems that support SNAPSHOT ISOLATION.)
- READ CONSISTENCY: This is a 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 `open`-ed. This is implemented by maintaining multiple versions of individual tuples, with a single transaction possibly referencing multiple versions of a single tuple. Modifications are maintained via long-term write locks, so when two transactions want to write the same object the first writer "wins", whereas in SNAPSHOT ISOLATION the first committer "wins".

Weak isolation schemes provide higher concurrency than serializability. As a result, some systems even use weak consistency as the default; Oracle defaults to READ COMMITTED, for example. 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 is tricky given the operationally-defined semantics of the schemes.

## 5.3 Log Manager

The log manager is responsible for maintaining the durability of committed transactions, and for facilitating the rollback of aborted transactions to ensure atomicity. It provides these features by maintaining a sequence of log records on disk, and a set of data structures in memory. In order to support correct behavior after crash, the memory-resident data structures obviously need to be re-createable from persistent data in the log and the database.

Database logging is an incredibly complex and detail-oriented topic. The canonical reference on database logging is the journal paper on ARIES [45], and a database expert should be familiar with the details of that paper. The ARIES paper not only explains its protocol, but also provides discussion of alternative design possibilities, and the problems that they can cause. This makes for dense but eventually rewarding reading. As a more digestible introduction, the Ramakrishnan/Gehrke textbook [53] provides a description of the basic ARIES protocol without side discussions or refinements, and we provide a set of powerpoint slides that accompany that discussion on our website (http://redbook.cs.berkeley.edu). Here we discuss some of the basic ideas in recovery, and try to explain the complexity gap between textbook and journal descriptions.

As is well known, 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 baroque 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, every commercial database system operates in a mode that Härder and Reuter call "DIRECT, STEAL/NOT-FORCE" [26]: (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 schedulers 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.

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 tradeoff is that the logic to redo and undo logical operations becomes quite involved, which can severely degrade performance during transaction abort and database recovery.[6] 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 performance is greatly enhanced by the presence of database *checkpoints* – consistent versions of the database from the recent past. A checkpoint limits the amount of log that the recovery process needs to consult and process. However, the naïve generation of checkpoints is too expensive to do during regular processing, so some 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.

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.

## 5.4 Locking and Logging in Indexes

Indexes are physical storage structures for accessing data in the database. The indexes themselves are invisible to database users, except inasmuch as they improve performance. Users cannot directly read or modify indexes, and hence user code need not be isolated (in the ACID sense) from changes to the index. This allows indexes to be managed via more efficient (and complex) transactional schemes than database data. The only invariant that index concurrency and recovery needs to preserve is that the index always returns transactionally-consistent tuples from the database.

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

### 5.4.1 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, which allow multiple transactions to access the same pages only if they can be guaranteed not to conflict in their use of a page's content. One such conflict is that a reading transaction wants to traverse a fully-packed internal page of the tree, and a concurrent inserting transaction is operating below that page, and hence might need to split it [4]. These conservative schemes sacrifice too much concurrency compared with the more recent ideas below.
- *Latch-coupling* schemes, in which 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 [46]. 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, which add some simple additional structure to the B+-tree to minimize the requirement for latches and retraversals. 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 rightlinks to find the new correct location in the tree. [39][35]

Kornacker, et al. [35] provide a detailed discussion of the distinctions between latch-coupling and right-link schemes, and points out that latch-coupling is only applicable to B+-trees, and will not work for index trees over more complex data, e.g. multidimensional indexes like R-trees.

### 5.4.2 Logging for Physical Structures

In addition to special-case concurrency logic, indexes employ special-case logging logic. This logic makes logging and recovery much more efficient, at the expense of more complexity in the code. The main idea is that structural index changes need not be undone when the associated transaction is aborted; such changes may 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, these changes should be left in place. ARIES provides an elegant mechanism for these scenarios called *nested top actions*, which allows the recovery process to "jump over" log records for physical structure modifications without any special case code during recovery.

This same idea is used in other contexts, including in heap files. An insertion into a heap file may require the file to be extended on disk. To capture this, changes must be made to the file's "extent map", 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 be in fact be useful for absorbing future insert traffic.

### 5.4.3 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 seriazability (including phantom protection) while allowing for tuple-level locks and the use of indexes.

The phantom problem arises 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 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. Unfortunately, it is well known that predicate locking is expensive, since it requires a way to compare arbitrary predicates for overlap – something that cannot be done with a hash-based lock table [2].

The standard solution 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* is required to allocate an exclusive lock on the "next-key" tuple that exists in the index: the tuple with 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 (e.g. if there were no 'Bob' on the 1st access, there should be no 'Bob' on subsequent accesses). One corner 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 to be modified as well, so that they must get a shared lock on the "next-key" tuple in the index as well:

the minimum-keyed tuple that does *not* satisfy the query predicate. An implementation of next-key locking is described for ARIES [42].

Next-key locking is not simply a clever hack. It is an instance 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 (e.g. hash-based lock tables) can be used for more complex purposes, simply by modifying the lock protocol. This idea of using physical surrogates for logical concepts is unique to database research: it is largely unappreciated in other systems work on concurrency, which typically does not consider semantic information about logical concepts as part of the systems challenge. Designers of complex software systems should keep this general approach in their "bag of tricks" when such semantic information is available.

## 5.5 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, and allow the implementation 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 complicating things further with access method details. Even with the simplification, things are deeply intertwined. One manifestation of the relationship between concurrency and recovery is that write-ahead logging makes implicit assumptions about the locking protocol – it 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 were previously modified by the transaction. 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 is unable to acquire the new locks it needs to complete the rollback process!

Access methods complicate things yet further. It is an enormous intellectual challenge to take a textbook access method (e.g. linear hashing [41] or R-trees [24]) and implement it correctly and efficiently in a transactional system; for this reason, most DBMSs still only implement heap files and B+-trees as native, transactionally protected access methods. 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 Multiversion concurrency control) do not usually consider access methods at all, or if they do mention them it is only in an offhanded and impractical fashion [36]. Hence it is unlikely that one can mix and match different concurrency mechanisms for a given access method implementation.

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, 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 (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 rightlink scheme for B+-trees assumes that pages in the tree never "re-merge" after they split, an assumption that requires the recovery scheme to use a scheme like 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 reimplement it as needed, e.g. the choice of pages to replace (because of the STEAL property), and the scheduling of page flushes (thanks to the NOT FORCE property). Of course achieving this isolation is the direct cause of much of the complexity in concurrency and recovery, so this spot is not perhaps as bright as it seems either.

# 6 Shared Components

In this section we cover a number of utility subsystems that are present in nearly all commercial DBMS, but rarely discussed in the literature.

## *6.1 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, and the correct management of this memory is both a programming burden and a performance issue. Selinger-style query optimization can use a great deal