



# Advanced Computer Systems (ACS)

DIKU Course Compendium

# **Advanced Computer Systems (ACS)**

DIKU Course Compendium

DIKU, Department of Computer Science,  
University of Copenhagen, Denmark

Block 2, 2015/16

# Contents

Preface	v
Learning Goals	vii
Source List	ix
1 Fundamental Abstractions	1
2 Modularity through Clients and Services, RPC	19
3 Techniques for Performance	27
4 Concurrency Control	45
5 Recovery	101
6 Experimental Design	127
7 Notions of Reliability	179
8 Topics in Distributed Coordination and Distributed Transactions	221
9 Communication and End-to-End Argument	317
10 Data Processing - External Sorting	383
11 Data Processing - Basic Relational Operators and Joins	395
12 Data Processing - Parallelism	459



# Preface

This compendium has been designed for the course Advanced Computer Systems (ACS), taught at the Department of Computer Science (DIKU), University of Copenhagen. The contents of the compendium are in correspondence with the rules of use at academic courses defined by CopyDan. The compendium is organized into 12 parts, each containing textbook chapters or reference papers related to topics covered in the course. Each part is also prefaced by a short description of the learning expectations with respect to the readings.

The compendium starts with a review of **fundamental abstractions** in computer systems, namely *interpreters*, *memory*, and *communication links* (Part 1). The course explores multiple **properties** that may be attached to these abstractions, and exposes principled design and implementation techniques to obtain these properties while respecting interfaces and achieving high performance. A first property is the notion of **strong modularity**, achieved by organization of interpreters into clients and services and use of remote procedure call (RPC) mechanics (Part 2).

After a brief review of general techniques for **performance** (Part 3), the properties of **atomicity** and **durability** are explored. The former property of atomicity may be understood with respect to before-or-after, or alternatively all-or-nothing semantics. Multiple different *concurrency control* protocols to achieve before-or-after atomicity over a memory abstraction are first introduced (Part 4). Following concurrency control, *recovery* protocols for all-or-nothing atomicity and durability are discussed (Part 5).

The text then ventures into a brief foray on techniques for **experimental design** (Part 6), which allow performance characteristics of different designs and implementations of a given abstraction to be analyzed. After this foray, the compendium then turns to the property of **high availability** in the presence of faults, achieved by a combination of techniques. First, general techniques for *reliability*, in particular replication techniques, are discussed (Part 7). Distribution of system functionality and replication introduce the problem of maintaining consistency. So, solutions for achieving high degrees of consistency in distributed scenarios, including *ordered multicast*, *two-phase commit*, and *state-machine replication*, are then discussed (Part 8). Finally, *communication* schemes that decouple system functions are discussed, along with the classic *end-to-end argument* (Part 9).

The text finally explores the property of **scalability** with large data volumes, and reviews design and implementation techniques for data processing operators, including *external sorting*, *basic relational operators and joins*, as well as *parallelism* (Parts 10, 11, and 12, respectively).

We hope you enjoy your readings!



# Learning Goals

The learning goals for ACS are listed below.

## **Knowledge**

- Describe the design of transactional and distributed systems, including techniques for modularity, performance, and fault tolerance.
- Explain how to employ strong modularity through a client-service abstraction as a paradigm to structure computer systems, while hiding complexity of implementation from clients.
- Explain techniques for large-scale data processing.

## **Skills**

- Implement systems that include mechanisms for modularity, atomicity, and fault tolerance.
- Structure and conduct experiments to evaluate a system's performance.

## **Competences**

- Discuss design alternatives for a modular computer system, identifying desired system properties as well as describing mechanisms for improving performance while arguing for their correctness.
- Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
- Apply principles of large-scale data processing to analyze concrete information-processing problems.



# Source List

- G. Coulouris, J. Dollimore, T. Kindberg. Distributed systems, concepts and design. Third Edition. Chapters 11 (except 11.2 and 11.3) and 13, pp. 419–423, 436–464, and 515–552 (72 of 772). Addison-Wesley, 2001. ISBN: 0201-61918-0
- J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. Commun. ACM 53, 1, pp. 72-77 (6 of 159), 2010. Doi: 10.1145/1629175.1629198
- D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. Commun. ACM 35, pp. 85-98 (14 of 1868), 1992. Doi: 10.1145/129888.129894
- D. Lilja. Measuring Computer Performance: A Practitioner's Guide. Chapters 1, 2, and 6, pp. 1–24 and 82–107 (50 of 261). Cambridge University Press, 2000. ISBN: 978-0-521-64105-0
- H. Garcia-Molina, J. D. Ullman, J. Widom. Database Systems: The Complete Book. Chapters 11.4 and 15, pp. 525–533 and 713–774 (71 of 1119). Prentice Hall, 2002. ISBN: 0-13-031995-3
- G. Graefe. Encapsulation of parallelism in the Volcano query processing system. SIGMOD Rec. 19, 2, pp. 102-111 (10 of 632), 1990. Doi: 10.1145/93605.98720
- D. Pritchett. BASE: An Acid Alternative. Queue 6, 3 (May 2008), pp. 48-55 (8 of 72), 2008. Doi=10.1145/1394127.1394128.
- R. Ramakrishnan and J. Gehrke. Database Management Systems. Third Edition. Chapters 16–18, pp. 519–544, 549–575, and 579–600 (75 of 1065). McGraw-Hill, 2003. ISBN: 978-0-07-246563-1
- J. H. Saltzer and M. F. Kaashoek. Principles of Computer System Design: An Introduction. Part I. Sections 2.1, 4.2, and 6.1, pp. 44–60, 167–172, and 300–316 (40 of 526). Morgan Kaufmann, 2009. ISBN: 978-0-12-374957-4
- J. H. Saltzer and M. F. Kaashoek. Principles of Computer System Design: An Introduction. Part II. Chapters 8.1–4 and 8.6, pp 8-2 – 8-35 and 8-51 – 8-54 (38 of 826). Creative Commons License, 2009.
- J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. ACM Trans. Comput. Syst. 2(4) pp. 277–288 (12 of 359), 1984. Doi: 10.1145/357401.357402

- F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22(4) pp. 299–319 (21 of 409), 1990. Doi: 10.1145/98163.98167
- A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Second Edition. Chapters 4, pp. 124–125 and 140–177 (40 of 686) Pearson International Edition, 2007. ISBN: 0-13-613553-6

## Chapter 1

# Fundamental Abstractions

This chapter contains the book chapter:

J. H. Saltzer and M. F. Kaashoek. Principles of Computer System Design: An Introduction. Part I. Section 2.1, pp. 44–60 (17 of 526). Morgan Kaufmann, 2009. ISBN: 978-0-12-374957-4

The chapter reviews the **fundamental abstractions** in computer systems: memory, interpreters, and communication links. These abstractions manifest themselves in hardware and in software, in centralized as well as distributed systems. *The ultimate goal of this portion of the material is to convey the generality of these abstractions, and stimulate us to reflect on how different versions of these abstractions are implemented in terms of one another over different system layers.*

The learning goals for this portion of the material are listed below.

- Identify the fundamental abstractions in computer systems and their APIs, including memory, interpreters, communication links.
- Explain how names are used in the fundamental abstractions.
- Design a top-level abstraction, respecting its correspondent API, based on lower-level abstractions.
- Discuss performance and fault-tolerance aspects of such a design.

---

**44 CHAPTER 2** Elements of Computer System Organization

2.5.12 The Shell and Implied Contexts, Search Paths, and Name Discovery .....	110
2.5.13 Suggestions for Further Reading .....	112
<b>Exercises .....</b>	<b>112</b>

---

## OVERVIEW

Although the number of potential abstractions for computer system components is unlimited, remarkably the vast majority that actually appear in practice fall into one of three well-defined classes: *the memory*, *the interpreter*, and *the communication link*. These three abstractions are so fundamental that theoreticians compare computer algorithms in terms of the number of data items they must remember, the number of steps their interpreter must execute, and the number of messages they must communicate.

Designers use these three abstractions to organize physical hardware structures, not because they are the only ways to interconnect gates, but rather because

- they supply fundamental functions of recall, processing, and communication,
- so far, these are the only hardware abstractions that have proven both to be widely useful and to have understandably simple interface semantics.

To meet the many requirements of different applications, system designers build layers on this fundamental base, but in doing so they do not routinely create completely different abstractions. Instead, they elaborate the same three abstractions, rearranging and repackaging them to create features that are useful and interfaces that are convenient for each application. Thus, for example, the designer of a general-purpose system such as a personal computer or a network server develops interfaces that exhibit highly refined forms of the same three abstractions. The user, in turn, may see the memory in the form of an organized file or database system, the interpreter in the form of a word processor, a game-playing system, or a high-level programming language, and the communication link in the form of instant messaging or the World Wide Web. On examination, underneath each of these abstractions is a series of layers built on the basic hardware versions of those same abstractions.

A primary method by which the abstract components of a computer system interact is *reference*. What that means is that the usual way for one component to connect to another is by *name*. Names appear in the interfaces of all three of the fundamental abstractions as well as the interfaces of their more elaborate higher-layer counterparts. The memory stores and retrieves objects by name, the interpreter manipulates named objects, and names identify communication links. Names are thus the glue that interconnects the abstractions. Named interconnections can, with proper design, be easy to change. Names also allow the sharing of objects, and they permit finding previously created objects at a later time.

This chapter briefly reviews the architecture and organization of computer systems in the light of abstraction, naming, and layering. Some parts of this review will be familiar to the reader with a background in computer software or hardware, but the systems perspective may provide some new insights into those familiar concepts and

it lays the foundation for coming chapters. [Section 2.1](#) describes the three fundamental abstractions, [Section 2.2](#) presents a model for naming and explains how names are used in computer systems, and [Section 2.3](#) discusses how a designer combines the abstractions, using names and layers, to create a typical computer system, presenting the file system as a concrete example of the use of naming and layering for the memory abstraction. [Section 2.4](#) looks at how the rest of this book will consist of designing some higher-level version of one or more of the three fundamental abstractions, using names for interconnection and built up in layers. [Section 2.5](#) is a case study showing how abstractions, naming, and layering are applied in a real file system.

## 2.1 THE THREE FUNDAMENTAL ABSTRACTIONS

We begin by examining, for each of the three fundamental abstractions, what the abstraction does, how it does it, its interfaces, and the ways it uses names for interconnection.

### 2.1.1 Memory

*Memory*, sometimes called *storage*, is the system component that remembers data values for use in computation. Although memory technology is wide-ranging, as suggested by the list of examples in [Figure 2.1](#), all memory devices fit a simple abstract model that has two operations, named `WRITE` and `READ`:

```
WRITE (name, value)
value ← READ (name)
```

The `WRITE` operation specifies in *value* a value to be remembered and in *name* a name by which one can recall that value in the future. The `READ` operation specifies in *name* the name of some previously remembered value, and the memory device returns that

value. A later call to `WRITE` that specifies the same name updates the value associated with that name.

Memories can be either volatile or non-volatile. A *volatile* memory is one whose mechanism of retaining information consumes energy; if its power supply is interrupted for some reason, it forgets its information content. When one turns off the power to a *non-volatile* memory (sometimes called “stable storage”), it retains its content, and when power is again available, `READ` operations return the same values as before. By connecting a volatile memory to a battery or an

#### Hardware memory devices:

- RAM chip
- Flash memory
- Magnetic tape
- Magnetic disk
- CD-R and DVD-R

#### Higher level memory systems:

- RAID
- File system
- Database management system

**FIGURE 2.1**

Some examples of memory devices that may be familiar.

**Sidebar 2.1 Terminology: Durability, Stability, and Persistence** Both in common English usage and in the professional literature, the terms *durability*, *stability*, and *persistence* overlap in various ways and are sometimes used almost interchangeably. In this text, we define and use them in a way that emphasizes certain distinctions.

*Durability* A property of a storage medium: the length of time it remembers.

*Stability* A property of an object: it is unchanging.

*Persistence* A property of an active agent: it keeps trying.

Thus, the current chapter suggests that files be placed in a durable storage medium—that is, they should survive system shutdown and remain intact for as long as they are needed. [Chapter 8](#) [on-line] revisits durability specifications and classifies applications according to their durability requirements.

This chapter introduces the concept of stable bindings for names, which, once determined, never again change.

[Chapter 7](#) [on-line] introduces the concept of a persistent sender, a participant in a message exchange who keeps retransmitting a message until it gets confirmation that the message was successfully received, and [Chapter 8](#) [on-line] describes persistent faults, which keep causing a system to fail.

uninterruptible power supply, it can be made *durable*, which means that it is designed to remember things for at least some specified period, known as its *durability*. Even non-volatile memory devices are subject to eventual deterioration, known as *decay*, so they usually also have a specified durability, perhaps measured in years. We will revisit durability in [Chapters 8](#) [on-line] and [10](#) [on-line], where we will see methods of obtaining different levels of durability. [Sidebar 2.1](#) compares the meaning of durability with two other, related words.

At the physical level, a memory system does not normally name, READ, or WRITE values of arbitrary size. Instead, hardware layer memory devices READ and WRITE contiguous arrays of bits, usually fixed in length, known by various terms such as *bytes* (usually 8 bits, but one sometimes encounters architectures with 6-, 7-, or 9-bit bytes), *words* (a small integer number of bytes, typically 2, 4, or 8), *lines* (several words), and *blocks* (a number of bytes, usually a power of 2, that can measure in the thousands). Whatever the size of the array, the unit of physical layer memory written or read is known as a memory (or storage) *cell*. In most cases, the *name* argument in the READ and WRITE calls is actually the name of a cell. Higher-layer memory systems also READ and WRITE contiguous arrays of bits, but these arrays usually can be of any convenient length, and are called by terms such as *record*, *segment*, or *file*.

### 2.1.1.1 Read/Write Coherence and Atomicity

Two useful properties for a memory are *read/write coherence* and *before-or-after atomicity*. Read/write coherence means that the result of the READ of a named cell is always the same as the most recent WRITE to that cell. Before-or-after atomicity

means that the result of every READ or WRITE is as if that READ or WRITE occurred either completely before or completely after any other READ or WRITE. Although it might seem that a designer should be able simply to assume these two properties, that assumption is risky and often wrong. There are a surprising number of threats to read/write coherence and before-or-after atomicity:

- *Concurrency.* In systems where different actors can perform READ and WRITE operations concurrently, they may initiate two such operations on the same named cell at about the same time. There needs to be some kind of arbitration that decides which one goes first and to ensure that one operation completes before the other begins.
- *Remote storage.* When the memory device is physically distant, the same concerns arise, but they are amplified by delays, which make the question of “which WRITE was most recent?” problematic and by additional forms of failure introduced by communication links. [Section 4.5](#) introduces remote storage, and [Chapter 10](#) [on-line] explores solutions to before-or-after atomicity and read/write coherence problems that arise with remote storage systems.
- *Performance enhancements.* Optimizing compilers and high-performance processors may rearrange the order of memory operations, possibly changing the very meaning of “the most recent WRITE to that cell” and thereby destroying read/write coherence for concurrent READ and WRITE operations. For example, a compiler might delay the WRITE operation implied by an assignment statement until the register holding the value to be written is needed for some other purpose. If someone else performs a READ of that variable, they may receive an old value. Some programming languages and high-performance processor architectures provide special programming directives to allow a programmer to restore read/write coherence on a case-by-case basis. For example, the Java language has a SYNCHRONIZED declaration that protects a block of code from read/write incoherence, and Hewlett-Packard’s Alpha processor architecture (among others) includes a *memory barrier* (MB) instruction that forces all preceding READS and WRITES to complete before going on to the next instruction. Unfortunately, both of these constructs create opportunities for programmers to make subtle mistakes.
- *Cell size incommensurate with value size.* A large value may occupy multiple memory cells, in which case before-or-after atomicity requires special attention. The problem is that both reading and writing of a multiple-cell value is usually done one cell at a time. A reader running concurrently with a writer that is updating the same multiple-cell value may end up with a mixed bag of cells, only some of which have been updated. Computer architects call this hazard *write tearing*. Failures that occur in the middle of writing multiple-cell values can further complicate the situation. To restore before-or-after atomicity, concurrent readers and writers must somehow be coordinated, and a failure in the middle of an update must leave either all or none of the intended update intact. When these conditions are met, the READ or WRITE is said to be *atomic*. A closely related

risk arises when a small value shares a memory cell with other small values. The risk is that if two writers concurrently update different values that share the same cell, one may overwrite the other's update. Atomicity can also solve this problem. [Chapter 5](#) begins the study of atomicity by exploring methods of coordinating concurrent activities. [Chapter 9](#) [on-line] expands the study of atomicity to also encompass failures.

- *Replicated storage.* As [Chapter 8](#) [on-line] will explore in detail, reliability of storage can be increased by making multiple copies of values and placing those copies in distinct storage cells. Storage may also be replicated for increased performance, so that several readers can operate concurrently. But replication increases the number of ways in which concurrent READ and WRITE operations can interact and possibly lose either read/write coherence or before-or-after atomicity. During the time it takes a writer to update several replicas, readers of an updated replica can get different answers from readers of a replica that the writer hasn't gotten to yet. [Chapter 10](#) [on-line] discusses techniques to ensure read/write coherence and before-or-after atomicity for replicated storage.

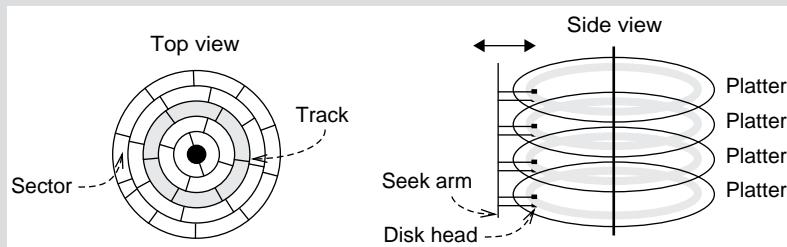
Often, the designer of a system must cope with not just one but several of these threats simultaneously. The combination of replication and remoteness is particularly challenging. It can be surprisingly difficult to design memories that are both efficient and also read/write coherent and atomic. To simplify the design or achieve higher performance, designers sometimes build memory systems that have weaker coherence specifications. For example, a multiple processor system might specify: "The result of a READ will be the value of the latest WRITE if that WRITE was performed by the same processor." There is an entire literature of "data consistency models" that explores the detailed properties of different memory coherence specifications. In a layered memory system, it is essential that the designer of a layer know precisely the coherence and atomicity specifications of any lower layer memory that it uses. In turn, if the layer being designed provides memory for higher layers, the designer must specify precisely these two properties that higher layers can expect and depend on. Unless otherwise mentioned, we will assume that physical memory devices provide read/write coherence for individual cells, but that before-or-after atomicity for multicell values (for example, files) is separately provided by the layer that implements them.

### 2.1.1.2 Memory Latency

An important property of a memory is the time it takes for a READ or a WRITE to complete, which is known as its *latency* (often called *access time*, though that term has a more precise definition that will be explained in [Sidebar 6.4](#)). In the magnetic disk memory (described in [Sidebar 2.2](#)) the latency of a particular sector depends on the mechanical state of the device at the instant the user requests access. Having read a sector, one may measure the time required to also read a different but nearby sector in microseconds—but only if the user anticipates the second read and requests it before the disk rotates past that second sector. A request just a few microseconds late may encounter

**Sidebar 2.2 How Magnetic Disks Work** Magnetic disks consist of rotating circular platters coated on both sides with a magnetic material such as ferric oxide. An electromagnet called a *disk head* records information by aligning the magnetic field of the particles in a small region on the platter's surface. The same disk head reads the data by sensing the polarity of the aligned particles as the platter spins by. The disk spins continuously at a constant rate, and the disk head actually floats just a few nanometers above the disk surface on an air cushion created by the rotation of the platter.

From a single position above a platter, a disk head can read or write a set of bits, called a *track*, located a constant distance from the center. In the top view below, the shaded region identifies a track. Tracks are formatted into equal-sized blocks, called *sectors*, by writing separation marks periodically around the track. Because all sectors are the same size, the outer tracks have more sectors than the inner ones.



A typical modern disk module, known as a “hard drive” because its platters are made of a rigid material, contains several platters spinning on a common axis called a *spindle*, as in the side view above. One disk head per platter surface is mounted on a comb-like structure that moves the heads in unison across the platters. Movement to a specific track is called *seeking*, and the comb-like structure is known as a *seek arm*. The set of tracks that can be read or written when the seek arm is in one position (for example, the shaded regions of the side view) is called a *cylinder*. Tracks, platters, and sectors are each numbered. A sector is thus addressed by geometric coordinates: track number, platter number, and rotational position. Modern disk controllers typically do the geometric mapping internally and present their clients with an address space consisting of consecutively numbered sectors.

To read or write a particular sector, the disk controller first seeks the desired track. Once the seek arm is in position, the controller waits for the beginning of the desired sector to rotate under the disk head, and then it activates the head on the desired platter. Physically encoding digital data in analog magnetic domains usually requires that the controller write complete sectors.

The time required for disk access is called *latency*, a term defined more precisely in Chapter 6. Moving a seek arm takes time. Vendors quote seek times of 5 to 10 milliseconds, but that is an average over all possible seek arm moves. A move from one

(Sidebar continues)

cylinder to the next may require only 1/20 of the time of a move from the innermost to the outermost track. It also takes time for a particular sector to rotate under the disk head. A typical disk rotation rate is 7200 rpm, for which the platter rotates once in 8.3 milliseconds. The time to transfer the data depends on the magnetic recording density, the rotation rate, the cylinder number (outer cylinders may transfer at higher rates), and the number of bits read or written. A platter that holds 40 gigabytes transfers data at rates between 300 and 600 megabits per second; thus a 1-kilobyte sector transfers in a microsecond or two. Seek time and rotation delay are limited by mechanical engineering considerations and tend to improve only slowly, but magnetic recording density depends on materials technology, which has improved both steadily and rapidly for many years.

Early disk systems stored between 20 and 80 megabytes. In the 1970s Kenneth Haughton, an IBM inventor, described a new technique of placing disk platters in a sealed enclosure to avoid contamination. The initial implementation stored 30 megabytes on each of two spindles, in a configuration known as a 30-30 drive. Haughton nicknamed it the “Winchester”, after the Winchester 30-30 rifle. The code name stuck, and for many years hard drives were known as Winchester drives. Over the years, Winchester drives have gotten physically smaller while simultaneously evolving to larger capacities.

a delay that is a thousand times longer, waiting for that second sector to again rotate under the read head. Thus the maximum rate at which one can transfer data to or from a disk is dramatically larger than the rate one would achieve when choosing sectors at random. A *random access memory (RAM)* is one for which the latency for memory cells chosen at random is approximately the same as the latency for cells chosen in the pattern best suited for that memory device. An electronic memory chip is usually configured for random access. Memory devices that involve mechanical movement, such as optical disks (CDs and DVDs) and magnetic tapes and disks, are not.

For devices that do not provide random access, it is usually a good idea, having paid the cost in delay of moving the mechanical components into position, to READ or WRITE a large block of data. Large-block READ and WRITE operations are sometimes relabeled GET and PUT, respectively, and this book uses that convention. Traditionally, the unqualified term *memory* meant random-access volatile memory and the term *storage* was used for non-volatile memory that is read and written in large blocks with GET and PUT. In practice, there are enough exceptions to this naming rule that the words “memory” and “storage” have become almost interchangeable.

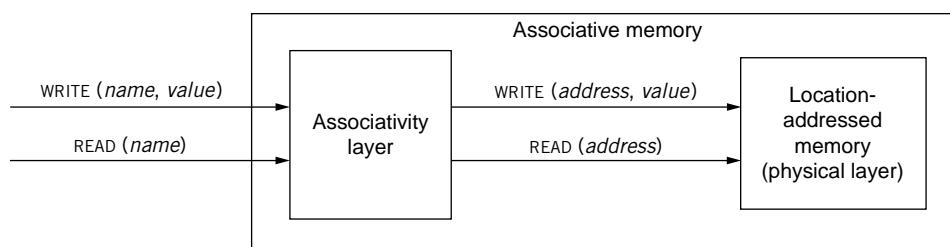
#### **2.1.1.3 Memory Names and Addresses**

Physical implementations of memory devices nearly always name a memory cell by the geometric coordinates of its physical storage location. Thus, for example, an electronic memory chip is organized as a two-dimensional array of flip-flops, each holding one named bit. The access mechanism splits the bit name into two parts, which in

turn go to a pair of multiplexers. One multiplexer selects an x-coordinate, the other a y-coordinate, and the two coordinates in turn select the particular flip-flop that holds that bit. Similarly, in a magnetic disk memory, one component of the name electrically selects one of the recording platters, while a distinct component of the name selects the position of the seek arm, thereby choosing a specific track on that platter. A third name component selects a particular sector on that track, which may be identified by counting sectors as they pass under the read head, starting from an index mark that identifies the first sector.

It is easy to design hardware that maps geometric coordinates to and from sets of names consisting of consecutive integers (0, 1, 2, etc.). These consecutive integer names are called *addresses*, and they form the *address space* of the memory device. A memory system that uses names that are sets of consecutive integers is called a *location-addressed memory*. Because the addresses are consecutive, the size of the memory cell that is named does not have to be the same as the size of the cell that is read or written. In some memory architectures each byte has a distinct address, but reads and writes can (and in some cases must always) occur in larger units, such as a word or a line.

For most applications, consecutive integers are not exactly the names that one would choose for recalling data. One would usually prefer to be allowed to choose less constrained names. A memory system that accepts unconstrained names is called an *associative memory*. Since physical memories are generally location-addressed, a designer creates an associative memory by interposing an associativity layer, which may be implemented either with hardware or software, that maps unconstrained higher-level names to the constrained integer names of an underlying location-addressed memory, as in [Figure 2.2](#). Examples of software associative memories, constructed on top of one or more underlying location-addressed memories, include personal telephone directories, file systems, and corporate database systems. A *cache*, a device that remembers the result of an expensive computation in the hope of not redoing that computation if it is needed again soon, is sometimes implemented as an



**FIGURE 2.2**

An associative memory implemented in two layers. The associativity layer maps the unconstrained names of its arguments to the consecutive integer addresses required by the physical layer location-addressed memory.

associative memory, either in software or hardware. (The design of caches is discussed in [Section 6.2](#).)

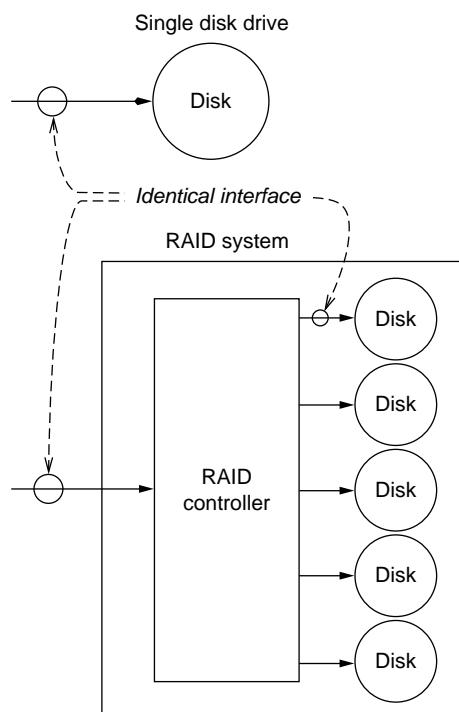
Layers that provide associativity and name mapping figure strongly in the design of all memory and storage systems. For example, [Table 2.2](#) on page 93 lists the layers of the [UNIX](#) file system. For another example of layering of memory abstractions, [Chapter 5](#) explains how memory can be virtualized by adding a name-mapping layer.

#### 2.1.1.4 Exploiting the Memory Abstraction: RAID

Returning to the subject of abstraction, a system known as RAID provides an illustration of the power of modularity and of how the storage abstraction can be applied to good effect. RAID is an acronym for Redundant Array of Independent (or Inexpensive) Disks. A RAID system consists of a set of disk drives and a controller configured with an electrical and programming interface that is identical to the interface of a single disk drive, as shown in [Figure 2.3](#). The RAID controller intercepts READ and WRITE requests

coming across its interface, and it directs them to one or more of the disks. RAID has two distinct goals:

- Improved performance, by reading or writing disks concurrently
- Improved durability, by writing information on more than one disk



**FIGURE 2.3**

Abstraction in RAID. The READ/WRITE electrical and programming interface of the RAID system, represented by the solid arrow, is identical to that of a single disk.

Different RAID configurations offer different trade-offs between these goals. Whatever trade-off the designer chooses, because the interface abstraction is that of a single disk, the programmer can take advantage of the improvements in performance and durability without reprogramming.

Certain useful RAID configurations are traditionally identified by (somewhat arbitrary) numbers. In later chapters, we will encounter several of these numbered configurations. The configuration known as RAID 0 (in [Section 6.1.5](#)) provides increased performance by allowing concurrent reading and writing. The configuration known as RAID 4 (shown in [Figure 8.6](#) [on-line]) improves disk reliability by applying error-correction codes. Yet another configuration known as RAID 1 (in [Section 8.5.4.6](#) [on-line]) provides high durability by

making identical copies of the data on different disks. Exercise 8.8 [on-line] explores a simple but elegant performance optimization known as RAID 5. These and several other RAID configurations were originally described in depth in a paper by Randy Katz, Garth Gibson, and David Patterson, who also assigned the traditional numbers to the different configurations [see Suggestions for Further Reading 10.2.2].

### 2.1.2 Interpreters

*Interpreters* are the active elements of a computer system; they perform the *actions* that constitute computations. Figure 2.4 lists some examples of interpreters that may be familiar. As with memory, interpreters also come in a wide range of physical manifestations. However, they too can be described with a simple abstraction, consisting of just three components:

1. An *instruction reference*, which tells the interpreter where to find its next instruction
2. A *repertoire*, which defines the set of actions the interpreter is prepared to perform when it retrieves an instruction from the location named by the instruction reference
3. An *environment reference*, which tells the interpreter where to find its *environment*, the current state on which the interpreter should perform the action of the current instruction

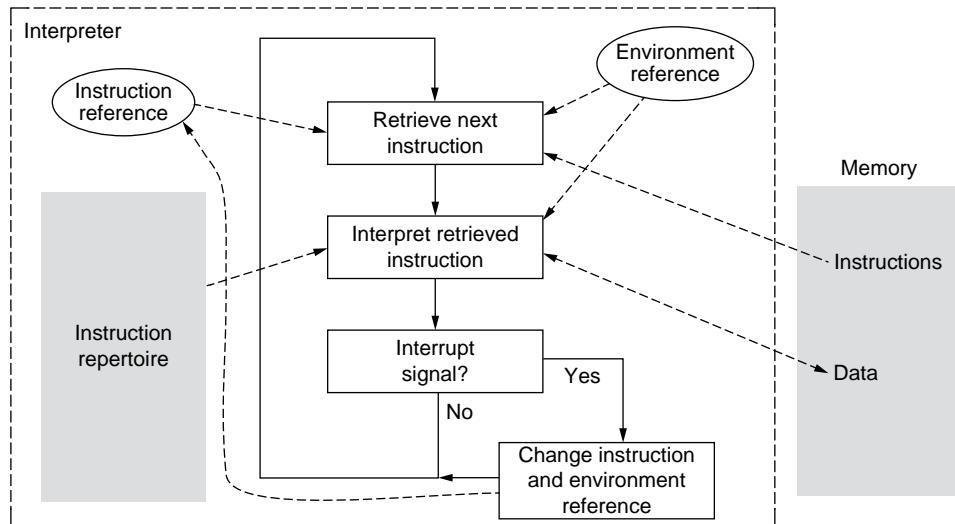
The normal operation of an interpreter is to proceed sequentially through some program, as suggested by the diagram and pseudocode of Figure 2.5. Using the environment reference to find the current environment, the interpreter retrieves from that environment the program instruction indicated in the instruction reference. Again using the environment reference, the interpreter performs the action directed by the

program instruction. That action typically involves using and perhaps changing data in the environment, and also an appropriate update of the instruction reference. When it finishes performing the instruction, the interpreter moves on, taking as its next instruction the one now named by the instruction reference. Certain events, called *interrupts*, may catch the attention of the interpreter, causing it, rather than the program, to supply the next instruction. The original program no longer controls the interpreter; instead, a different program, the interrupt handler, takes control and handles the event. The interpreter may also change the environment reference to one that is appropriate for the interrupt handler.

Hardware:
Pentium 4, PowerPC 970, UltraSPARC T1
disk controller
display controller
Software:
Alice, AppleScript, Perl, Tcl, Scheme
LISP, Python, Forth, Java bytecode
JavaScript, Smalltalk
TeX, LaTeX
Safari, Internet Explorer, Firefox

**FIGURE 2.4**

Some common examples of interpreters. The disk controller example is explained in Section 2.3 and the Web browser examples are the subject of Exercise 4.5.



```

1 procedure INTERPRET()
2   do forever
3     instruction ← READ (instruction_reference)
4     perform instruction in the context of environment_reference
5     if interrupt_signal = TRUE then
6       instruction_reference ← entry point of INTERRUPT_HANDLER
7       environment_reference ← environment ref of INTERRUPT_HANDLER
  
```

**FIGURE 2.5**

Structure of, and pseudocode for, an abstract interpreter. Solid arrows show control flow, and dashed arrows suggest information flow. Sidebar 2.3 describes this book's conventions for expressing pseudocode.

**Sidebar 2.3 Representation: Pseudocode and Messages** This book presents many examples of program fragments. Most of them are represented in pseudocode, an imaginary programming language that adopts familiar features from different existing programming languages as needed and that occasionally intersperses English text to characterize some step whose exact detail is unimportant. The pseudocode has some standard features, several of which this brief example shows.

```

1 procedure SUM (a, b) // Add two numbers.
2   total ← a + b
3   return total
  
```

The line numbers on the left are not part of the pseudocode; they are there simply to allow the text to refer to lines in the program. Procedures are explicitly declared  
*(Sidebar continues)*

(as in line 1), and indentation groups blocks of statements together. Program variables are set in *italic*, program key words in **bold**, and literals such as the names of procedures and built-in constants in SMALL CAPS. The left arrow denotes substitution or assignment (line 2) and the symbol "=" denotes equality in conditional expressions. The double slash precedes comments that are not part of the pseudocode. Various forms of iteration (**while**, **until**, **for each**, **do occasionally**), conditionals (**if**), set operations (**is in**), and case statements (**do case**) appear when they are helpful in expressing an example. The construction **for** *j* **from** 0 **to** 3 iterates four times; array indices start at 0 unless otherwise mentioned. The construction *y.x* means the element named *x* in the structure named *y*. To minimize clutter, the pseudocode omits declarations wherever the meaning is reasonably apparent from the context. Procedure parameters are passed by value unless the declaration **reference** appears. Section 2.2.1 of this chapter discusses the distinction between use by value and use by reference. When more than one variable uses the same structure, the declaration *structure\_name instance variable\_name* may be used.

The notation *a*(11...15) denotes extraction of bits 11 through 15 from the string *a* (or from the variable *a* considered as a string). Bits are numbered left to right starting with zero, with the most significant bit of integers first (using big-endian notation, as described in Sidebar 4.3). The + operator, when applied to strings, concatenates the strings.

Some examples are represented in the instruction repertoire of an imaginary reduced instruction set computer (RISC). Because such programs are cumbersome, they appear only when it is essential to show how software interacts with hardware.

In describing and using communication links, the notation

$$x \Rightarrow y: \{M\}$$

represents a message with contents *M* from sender *x* to recipient *y*. The notation {*a*, *b*, *c*} represents a message that contains the three named fields marshaled in some way that the recipient presumably understands how to unmarshal.

Many systems have more than one interpreter. Multiple interpreters are usually *asynchronous*, which means that they run on separate, uncoordinated, clocks. As a result, they may progress at different rates, even if they are nominally identical and running the same program. In designing algorithms that coordinate the work of multiple interpreters, one usually assumes that there is no fixed relation among their progress rates and therefore that there is no way to predict the relative timing, for example, of the LOAD and STORE instructions that they issue. The assumption of interpreter asynchrony is one of the reasons memory read/write coherence and before-or-after atomicity can be challenging design problems.

### 2.1.2.1 Processors

A general-purpose processor is an implementation of an interpreter. For purposes of concrete discussion throughout this book, we use a typical reduced instruction set processor. The processor's instruction reference is a *program counter*, stored in a fast memory register inside the processor. The program counter contains the address of the memory location that stores the next instruction of the current program. The environment reference of the processor consists in part of a small amount of built-in location-addressed memory in the form of named (by number) registers for fast access to temporary results of computations.

Our general-purpose processor may be directly wired to a memory, which is also part of its environment. The addresses in the program counter and in instructions are then names in the address space of that memory, so this part of the environment reference is wired in and unchangeable. When we discuss virtualization in [Chapter 5](#), we will extend the processor to refer to memory indirectly via one or more registers. With that change, the environment reference is maintained in those registers, thus allowing addresses issued by the processor to map to different names in the address space of the memory.

The repertoire of our general-purpose processor includes instructions for expressing computations such as adding two numbers (`ADD`), subtracting one number from another (`SUB`), comparing two numbers (`CMP`), and changing the program counter to the address of another instruction (`JMP`). These instructions operate on values stored in the named registers of the processor, which is why they are colloquially called “op-codes”.

The repertoire also includes instructions to move data between processor registers and memory. To distinguish program instructions from memory operations, we use the name `LOAD` for the instruction that `READS` a value from a named memory cell into a register of the processor and `STORE` for the instruction that `WRITES` the value from a register into a named memory cell. These instructions take two integer arguments, the name of a memory cell and the name of a processor register.

The general-purpose processor provides a *stack*, a push-down data structure that is stored in memory and used to implement procedure calls. When calling a procedure, the caller pushes arguments of the called procedure (the callee) on the stack. When the callee returns, the caller pops the stack back to its previous size. This implementation of procedures supports recursive calls because every invocation of a procedure always finds its arguments at the top of the stack. We dedicate one register for implementing stack operations efficiently. This register, known as the *stack pointer*, holds the memory address of the top of the stack.

As part of interpreting an instruction, the processor increments the program counter so that, when that instruction is complete, the program counter contains the address of the next instruction of the program. If the instruction being interpreted is a `JMP`, that instruction loads a new value into the program counter. In both cases, the flow of instruction interpretation is under control of the running program.

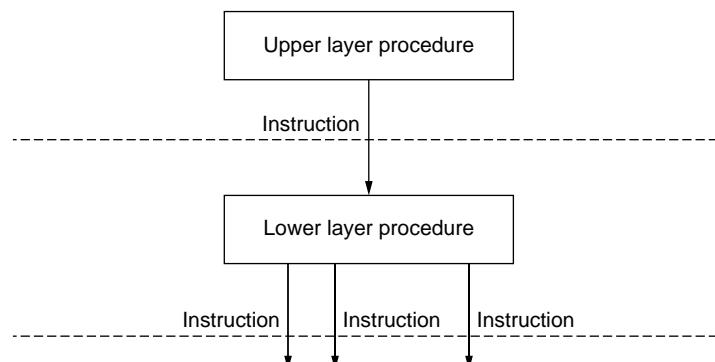
The processor also implements interrupts. An interrupt can occur because the processor has detected some problem with the running program (e.g., the program attempted to execute an instruction that the interpreter does not or cannot

implement, such as dividing by zero). An interrupt can also occur because a signal arrives from outside the processor, indicating that some external device needs attention (e.g., the keyboard signals that a key press is available). In the first case, the interrupt mechanism may transfer control to an *exception* handler elsewhere in the program. In the second case, the interrupt handler may do some work and then return control to the original program. We shall return to the subject of interrupts and the distinction between interrupt handlers and exception handlers in the discussion of threads in [Chapter 5](#).

In addition to general-purpose processors, computer systems typically also have special-purpose processors, which have a limited repertoire. For example, a clock chip is a simple, hard-wired interpreter that just counts: at some specified frequency, it executes an ADD instruction, which adds 1 to the contents of a register or memory location that corresponds to the clock. All processors, whether general-purpose or specialized, are examples of interpreters. However, they may differ substantially in the repertoire they provide. One must consult the device manufacturer's manual to learn the repertoire.

### 2.1.2.2 Interpreter Layers

Interpreters are nearly always organized in layers. The lowest layer is usually a hardware engine that has a fairly primitive repertoire of instructions, and successive layers provide an increasingly rich or specialized repertoire. A full-blown application system may involve four or five distinct layers of interpretation. Across any given layer interface, the lower layer presents some repertoire of possible instructions to the upper layer. [Figure 2.6](#) illustrates this model.



**FIGURE 2.6**

The model for a layered interpreter. Each layer interface, shown as a dashed line, represents an abstraction barrier, across which an upper layer procedure requests execution of instructions from the repertoire of the lower layer. The lower layer procedure typically implements an instruction by performing several instructions from the repertoire of a next lower layer interface.

---

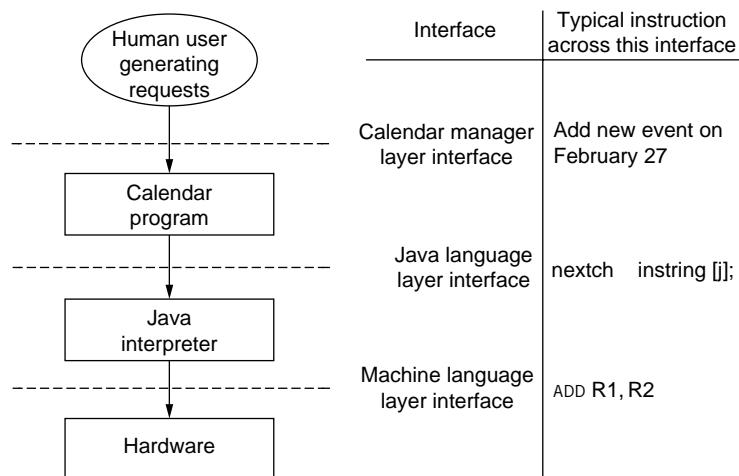
## 58 CHAPTER 2 Elements of Computer System Organization

Consider, for example, a calendar management program. The person making requests by moving and clicking a mouse views the calendar program as an interpreter of the mouse gestures. The instruction reference tells the interpreter to obtain its next instruction from the keyboard and mouse. The repertoire of instructions is the set of available requests—to add a new event, to insert some descriptive text, to change the hour, or to print a list of the day's events. The environment is a set of files that remembers the calendar from day to day.

The calendar program implements each action requested by the user by invoking statements in some programming language such as Java. These statements—such as iteration statements, conditional statements, substitution statements, procedure calls—constitute the instruction repertoire of the next lower layer. The instruction reference keeps track of which statement is to be executed next, and the environment is the collection of named variables used by the program. (We are assuming here that the Java language program has not been compiled directly to machine language. If a compiler is used, there would be one less layer.)

The actions of the programming language are in turn implemented by hardware machine language instructions of some general-purpose processor, with its own instruction reference, repertoire, and environment reference.

Figure 2.7 illustrates the three layers just described. In practice, the layered structure may be deeper—the calendar program is likely to be organized with an internal upper layer that interprets the graphical gestures and a lower layer that manipulates the calendar data, the Java interpreter may have an intermediate byte-code interpreter layer, and some machine languages are implemented with a microcode interpreter layer on top of a layer of hardware gates.



**FIGURE 2.7**

An application system that has three layers of interpretation, each with its own repertoire of instructions.

One goal in the design of a layered interpreter is to ensure that the designer of each layer can be confident that the layer below either completes each instruction successfully or does nothing at all. Half-finished instructions should never be a concern, even if there is a catastrophic failure. That goal is another example of atomicity, and achieving it is relatively difficult. For the moment, we simply assume that interpreters are atomic, and we defer the discussion of how to achieve atomicity to [Chapter 9](#) [on-line].

### 2.1.3 Communication Links

A *communication link* provides a way for information to move between physically separated components. Communication links, of which a few examples are listed in [Figure 2.8](#), come in a wide range of technologies, but, like memories and interpreters, they can be described with a simple abstraction. The communication link abstraction has two operations:

```
SEND (link_name, outgoing_message_buffer)
RECEIVE (link_name, incoming_message_buffer)
```

The SEND operation specifies an array of bits, called a *message*, to be sent over the communication link identified by *link\_name* (for example, a wire). The argument *outgoing\_message\_buffer* identifies the message to be sent, usually by giving the address and size of a buffer in memory that contains the message. The RECEIVE operation accepts an incoming message, again usually by designating the address and size of a buffer in memory to hold the incoming message. Once the lowest layer of a system has received a message, higher layers may acquire the message by calling a RECEIVE interface of the lower layer, or the lower layer may “upcall” to the higher layer, in which case the interface might be better characterized as DELIVER (*incoming\_message*).

Names connect systems to communication links in two different ways. First, the *link\_name* arguments of SEND and RECEIVE identify one of possibly several available communication links attached to the system.

Second, some communication links are actually multiply-attached networks of links, and some additional method is needed to name which of several possible recipients should receive the message. The name of the intended recipient is typically one of the components of the message.

At first glance, it might appear that sending and receiving a message is just an example of copying an array of bits from one memory to another memory over a wire using a sequence of READ and WRITE operations,

#### Hardware technology:

twisted pair

coaxial cable

optical fiber

#### Higher level

Ethernet

Universal Serial Bus (USB)

the Internet

the telephone system

a UNIX pipe

**FIGURE 2.8**

Some examples of communication links.

so there is no need for a third abstraction. However, communication links involve more than simple copying—they have many complications, such as a wide range of operating parameters that makes the time to complete a `SEND` or `RECEIVE` operation unpredictable, a hostile environment that threatens integrity of the data transfer, asynchronous operation that leads to the arrival of messages whose size and time of delivery can not be known in advance, and most significant, the message may not even be delivered. Because of these complications, the semantics of `SEND` and `RECEIVE` are typically quite different from those associated with `READ` and `WRITE`. Programs that invoke `SEND` and `RECEIVE` must take these different semantics explicitly into account. On the other hand, some communication link implementations do provide a layer that does its best to hide a `SEND/RECEIVE` interface behind a `READ/WRITE` interface.

Just as with memory and interpreters, designers organize and implement communication links in layers. Rather than continuing a detailed discussion of communication links here, we defer that discussion to [Section 7.2 \[on-line\]](#), which describes a three-layer model that organizes communication links into systems called *networks*. [Figure 7.18 \[on-line\]](#) illustrates this three-layer network model, which comprises a link layer, a network layer, and an end-to-end layer.

---

## 2.2 NAMING IN COMPUTER SYSTEMS

Computer systems use names in many ways in their construction, configuration, and operation. The previous section mentioned memory addresses, processor registers, and link names, and [Figure 2.9](#) lists several additional examples, some of which are probably familiar, others of which will turn up in later chapters. Some system names resemble those of a programming language, whereas others are quite different. When building systems out of subsystems, it is essential to be able to use a subsystem without having to know details of how that subsystem refers to its components. Names are thus used to achieve modularity, and at the same time, modularity must sometimes hide names.

We approach names from an object point of view: the computer system manipulates *objects*. An interpreter performs the manipulation under control of a program or perhaps under the direction of a human user. An object may be structured, which means that it uses other objects as components. In a direct analogy with two ways in which procedures can pass arguments, there are two ways to arrange for one object to use another as a component:

- create a copy of the component object and include the copy in the using object (use by *value*), or
- choose a name for the component object and include just that name in the using object (use by *reference*). The component object is said to *export* the name.

When passing arguments to procedures, use by value enhances modularity, because if the callee accidentally modifies the argument it does not affect the original. But use by value can be problematic because it does not easily permit two or more objects to *share* a component object whose value changes. If both object A

## Chapter 2

# Modularity through Clients and Services, RPC

This chapter contains the book chapter:

J. Saltzer and M. F. Kaashoek. Principles of Computer System Design: An Introduction. Part I. Section 4.2, pp. 167–172 (6 of 526). Morgan Kaufmann, 2009. ISBN: 978-0-12-374957-4

The chapter discusses how to organize interpreters in terms of clients and services. An important property of this organization is **strong modularity**, the capability of bounding failure propagation between these components. A classic mechanism to achieve strong modularity with clients and services is the remote procedure call (RPC). *The ultimate goal of this portion of the material is to enable us to write strongly modular software with RPCs, and to reflect on how strong modularity affects program semantics but at the same time allows us to incorporate additional mechanisms (e.g., for scaling the number of connections) in-between modular clients and services.*

The learning goals for this portion of the material are listed below.

- Recognize and explain modular designs with clients and services.
- Predict the functioning of service calls under different RPC semantics and failure modes.
- Identify different mechanisms to achieve RPCs.
- Implement RPC services with an appropriate mechanism, such as web services.

Note that the course assignments will be instrumental in achieving the goals in this as well as other parts of the course.

*This page has intentionally been left blank.*

---

## 4.2 COMMUNICATION BETWEEN CLIENT AND SERVICE

This section describes two extensions to sending and receiving messages. First, it introduces *remote procedure call (RPC)*, a stylized form of client/service interaction in which each request is followed by a response. The goal of RPC systems is to make a remote procedure call look like an ordinary procedure call. Because a service fails independently from a client, however, a remote procedure call can generally not offer identical semantics to procedure calls. As explained in the next subsection, some RPC systems provide various alternative semantics and the programmer must be aware of the details.

Second, in some applications it is desirable to be able to send messages to a recipient that is not on-line and to receive messages from a sender that is not on-line. For example, electronic mail allows users to send e-mail without requiring the recipient to be on-line. Using an intermediary for communication, we can implement these applications.

### 4.2.1 Remote Procedure Call (RPC)

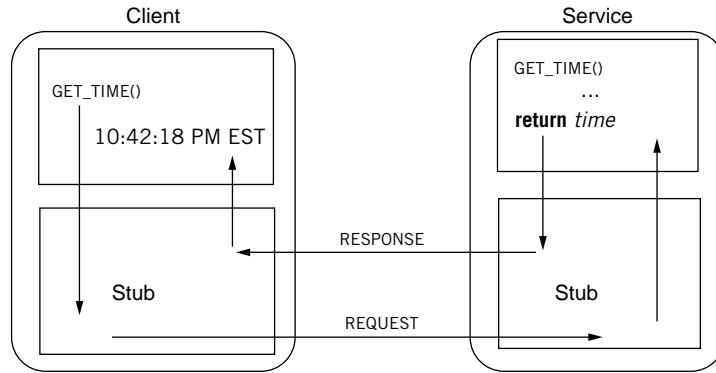
In many of the examples in the previous section, the client and service interact in a stylized fashion: the client sends a request, and the service replies with a response after processing the client's request. This style is so common that it has received its own name: *remote procedure call*, or RPC for short.

RPCs come in many varieties, adding features to the basic request/response style of interaction. Some RPC systems, for example, simplify the programming of clients and services by hiding many the details of constructing and formatting messages. In the time service example above, the programmer must call `SEND_MESSAGE` and `RECEIVE_MESSAGE`, and convert results into numbers, and so on. Similarly, in the file service example, the client and service have to construct messages and convert numbers into bit strings and the like. Programming these conversions is tedious and error prone.

*Stubs* remove this burden from the programmer (see [Figure 4.7](#)). A stub is a procedure that hides the marshaling and communication details from the caller and callee. An RPC system can use stubs as follows. The client module invokes a remote procedure, say `GET_TIME`, in the same way that it would call any other procedure. However, `GET_TIME` is actually just the name of a stub procedure that runs inside the client module (see [Figure 4.8](#)). The stub marshals the arguments of a call into a message, sends the message, and waits for a response. On arrival of the response, the client stub unmarshals the response and returns to the caller.

Similarly, a service stub waits for a message, unmarshals the arguments, and calls the procedure that the client requests (`GET_TIME` in the example). After the procedure returns, the service stub marshals the results of the procedure call into a message and sends it in a response to the client stub.

Writing stubs that convert more complex objects into an appropriate on-wire representation becomes quite tedious. Some high-level programming languages

**FIGURE 4.7**

Implementation of a remote procedure call using stubs. The stubs hide all remote communication from the caller and callee.

```
Client program
1  procedure MEASURE (func)
2      start ← GET_TIME (SECONDS)
3      func ()      // invoke the function
4      end ← GET_TIME (SECONDS)
5      return end – start
6
7  procedure GET_TIME (unit)      // the client stub for GET_TIME
8      SEND_MESSAGE (NameForTimeService, {"Get time", unit})
9      response ← RECEIVE_MESSAGE (NameForClient)
10     return CONVERT2INTERNAL (response)

Service program
1  procedure TIME_SERVICE ()      // the service stub for GET_TIME
2  do forever
3      request ← RECEIVE_MESSAGE (NameForTimeService)
4      opcode ← GET_OPCODE (request)
5      unit ← GET_ARGUMENT (request)
6      if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
7          response ← {"ok", GET_TIME (unit)}
8      else
9          response ← {"Bad request"}
10     SEND_MESSAGE (NameForClient, response)
```

**FIGURE 4.8**

GET\_TIME client and service using stubs.

such as Java can generate these stubs automatically from an interface specification [Suggestions for Further Reading 4.1.3], simplifying client/service programming even further. Figure 4.9 shows the client for such an RPC system. The RPC system would generate a procedure similar to the GET\_TIME stub in Figure 4.8. The client program of Figure 4.9 looks almost identical to the one using a local procedure call on page 149,

*The client program*

```

1   procedure MEASURE (func)
2     try
3       start  $\leftarrow$  GET_TIME (SECONDS)
4     catch (signal) servicefailed
5       return servicefailed
6     func () // invoke the function
7     try
8       end  $\leftarrow$  GET_TIME (SECONDS)
9     catch (signal) servicefailed
10    return servicefailed
11    return end - start

```

**FIGURE 4.9**

GET\_TIME client using a system that generates RPC stubs automatically.

except that it handles an additional error because remote procedure calls are not identical to procedure calls (as discussed below). The procedure that the service calls on line 7 is just the original procedure GET\_TIME on page 149.

Whether a system uses RPC with automatic stub generation is up to the implementers. For example, some implementations of Sun's Network File System (see Section 4.5) use automatic stub generation, but others do not.

### 4.2.2 RPCs are not Identical to Procedure Calls

It is tempting to think that by using stubs one can make a remote procedure call behave exactly the same as an ordinary procedure call, so that a programmer doesn't have to think about whether the procedure runs locally or remotely. In fact, this goal was a primary one when RPC was originally proposed—hence the name remote “procedure call”. However, RPCs are different from ordinary procedure calls in three important ways: First, RPCs can reduce fate sharing between caller and callee by exposing the failures of the callee to the caller so that the caller can recover. Second, RPCs introduce new failures that don't appear in procedure calls. These two differences change the semantics of remote procedure calls as compared with ordinary procedure calls, and the changes usually require the programmer to make adjustments to the surrounding code. Third, remote procedure calls take more time than procedure calls; the number of instructions to invoke a procedure (see Figure 4.2) is much less than the cost of invoking a stub, marshaling arguments, sending a request over a network, invoking a service stub, unmarshaling arguments, marshaling the response, receiving the response over the network, and unmarshaling the response.

To illustrate the first difference, consider writing a procedure call to the library program SQRT, which computes the square root of its argument *x*. A careful programmer would plan for the case that SQRT (*x*) will fail when *x* is negative by providing an explicit exception handler for that case. However, the programmer using ordinary procedure calls almost certainly doesn't go to the trouble of planning for certain possible failures because they have negligible probability. For

example, the programmer probably would not think of setting an interval timer when invoking `SQRT(x)`, even though `SQRT` internally has a successive-approximation loop that, if programmed wrong, might not terminate.

But now consider calling `SQRT` with an RPC. An interval timer suddenly becomes essential because the network between client and service can lose a message, or the other computer can crash independently. To avoid fate sharing, the RPC programmer must adjust the code to prepare for and handle this failure. When the client receives a “service failure” signal, the client may be able to recover by, for example, trying a different service or choosing an alternative algorithm that doesn’t use a remote service.

The second difference between ordinary procedure calls and RPCs is that RPCs introduce a new failure mode, the “no response” failure. When there is no response from a service, the client cannot tell which of two things went wrong: (1) some failure occurred before the service had a chance to perform the requested action, or (2) the service performed the action and then a failure occurred, causing just the response to be lost.

Most RPC designs handle the no-response case by choosing one of three implementation strategies:

- *At-least-once* RPC. If the client stub doesn’t receive a response within some specific time, the stub resends the request as many times as necessary until it receives a response from the service. This implementation may cause the service to execute a request more than once. For applications that call `SQRT`, executing the request more than once is harmless because with the same argument `SQRT` should always produce the same answer. In programming language terms, the `SQRT` service has no side effects. Such side-effect-free operations are also *idempotent*: repeating the same request or sequence of requests several times has the same effect as doing it just once. An at-least-once implementation does not provide the guarantee implied by its name. For example, if the service was located in a building that has been blown away by a hurricane, retrying doesn’t help. To handle such cases, an at-least-once RPC implementation will give up after some number of retries. When that happens, the request may have been executed more than once or not at all.
- *At-most-once* RPC. If the client stub doesn’t receive a response within some specific time, then the client stub returns an error to the caller, indicating that the service may or may not have processed the request. At-most-once semantics may be more appropriate for requests that do have side effects. For example, in a banking application, using at-least-once semantics for a request to transfer \$100 from one account to another could result in multiple \$100 transfers. Using at-most-once semantics assures that either zero or one transfers take place, a somewhat more controlled outcome. Implementing at-most-once RPC is harder than it sounds because the underlying network may duplicate the request message without the client stub’s knowledge. Chapter 7 [on-line] describes an at-most-once implementation, and Birrell and Nelson’s paper gives

a nice, complete description of an RPC system that implements at-most-once [Suggestions for Further Reading 4.1.1].

- *Exactly-once* RPC. These semantics are the ideal, but because the client and service are independent it is in principle impossible to guarantee. As in the case of at-least-once, if the service is in a building that was blown away by a hurricane, the best the client stub can do is return error status. On the other hand, by adding the complexity of extra message exchanges and careful record-keeping, one can approach exactly-once semantics closely enough to satisfy some applications. The general idea is that, if the RPC requesting transfer of \$100 from account A to B produces a “no response” failure, the client stub sends a separate RPC request to the service to ask about the status of the request that got no response. This solution requires that both the client and the service stubs keep careful records of each remote procedure call request and response. These records must be fault tolerant because the computer running the service might fail and lose its state between the original RPC and the inquiry to check on the RPC’s status. Chapters 8 [on-line] through 10 [on-line] introduce the necessary techniques.

The programmer must be aware that RPC semantics differ from those of ordinary procedure calls, and because different RPC systems handle the no-response case in different ways, it is important to understand just which semantics any particular RPC system tries to provide. Even if the name of the implementation implies a guarantee (e.g., at-least-once), we have seen that there are cases in which the implementation cannot deliver it. One cannot simply take a collection of legacy programs and arbitrarily separate the modules with RPC. Some thought and reprogramming is inevitably required. Problem set 2 explores the effects of different RPC semantics in the context of a simple client/service application.

The third difference is that calling a local procedure takes typically much less time than calling a remote procedure call. For example, invoking a remote `SQRT` is likely to be more expensive than the computation for `SQRT` itself because the overhead of a remote procedure call is much higher than the overhead of following the procedure calling conventions. To hide the cost of a remote procedure call, a client stub may deploy various performance-enhancing techniques (see Chapter 6), such as caching results and pipelining requests (as is done in the X Window System of Sidebar 4.4). These techniques increase complexity and can introduce new problems (e.g., how to ensure that the cache at the client stays consistent with the one at the service). The performance difference between procedure calls and remote procedure calls requires the designer to consider carefully what procedure calls should be remote ones and which ones should be ordinary, local procedure calls.

A final difference between procedure calls and RPCs is that some programming language features don’t combine well with RPC. For example, a procedure that communicates with another procedure through global variables cannot typically be executed remotely because separate computers usually have separate address spaces. Similarly, other language constructs that use explicit addresses won’t work. Arguments

consisting of data structures that contain pointers, for example, are a problem because pointers to objects in the client computer are local addresses that have different bindings when resolved in the service computer. It is possible to design systems that use global references for objects that are passed by reference to remote procedure calls but require significant additional machinery and introduce new problems. For example, a new plan is needed for determining whether an object can be deleted locally because a remote computer might still have a reference to the object. Solutions exist, however; see, for example, the article on Network Objects [Suggestions for Further Reading 4.1.2].

Since RPCs don't provide the same semantics as procedure calls, the word "procedure" in "remote procedure call" can be misleading. Over the years the concept of RPC has evolved from its original interpretation as an exact simulation of an ordinary procedure call to instead mean any client/service interaction in which the request is followed by a response. This text uses this modern interpretation.

#### 4.2.3 Communicating through an Intermediary

Sending a message from a sender to a receiver requires that both parties be available at the same time. In many applications this requirement is too strict. For example, in electronic mail we desire that a user be able to send an e-mail to a recipient even if the recipient is not on-line at the time. The sender sends the message and the recipient receives the message some time later, perhaps when the sender is not on-line. We can implement such applications using an intermediary. In the case of communication, this intermediary doesn't have to be trusted because communication applications often consider the intermediary to be part of an untrusted network and have a separate plan for securing messages (as we will see in Chapter 11 [on-line]).

The primary purpose of the e-mail intermediary is to implement *buffered* communication. Buffered communication provides the SEND/RECEIVE abstraction but avoids the requirement that the sender and receiver be present simultaneously. It allows the delivery of a message to be shifted in time. The intermediary can hold messages until the recipient comes on-line. The intermediary might buffer messages in volatile memory or in non-volatile memory, such as a file system. The latter design allows the intermediary to buffer messages across power failures.

Once we have an intermediary, three interesting design opportunities arise. First, the sender and receiver may make different choices of whether to *push* or *pull* messages. Push is when the initiator of a data movement sends the data. Pull is when the initiator of a data movement asks the other end to send it the data. These definitions are independent of whether or not the system uses an intermediary, but in systems with intermediaries it is not uncommon to find both in a single system. For example, the sender in the Internet's e-mail system, Simple Mail Transfer Protocol (SMTP), pushes the mail to the service that holds the recipient's mailbox. On the other hand, the receiving client pulls messages to fetch mail from a mailbox: the user hits the

## Chapter 3

# Techniques for Performance

This chapter contains the book chapter:

J. Saltzer and M. F. Kaashoek. Principles of Computer System Design: An Introduction. Part I. Section 6.1, pp. 300–316 (17 of 526). Morgan Kaufmann, 2009. ISBN: 978-0-12-374957-4

An important consequence of writing strongly modular software with clients and services is that service implementations may be optimized for performance or for reliability (or both) without affecting clients, as long as these optimizations do not affect semantics. **Performance** is a recurring theme in design and implementation of system abstractions and services. There are a number of general techniques for performance, among which *concurrency* is of special importance. *The ultimate goal of this portion of the material is to introduce us to metrics to characterize performance, and provide us with general techniques to design services for performance.*

The learning goals for this portion of the material are listed below.

- Explain performance metrics such as latency, throughput, overhead, utilization, capacity, and scalability.
- List common hardware parameters that affect performance.
- Apply performance improvement techniques, such as concurrency, batching, dallying, and fast-path coding.

---

## OVERVIEW

The specification of a computer system typically includes explicit (or implicit) performance goals. For example, the specification may indicate how many concurrent users the system should be able to support. Typically, the simplest design fails to meet these goals because the design has a *bottleneck*, a stage in the computer system that takes longer to perform its task than any of the other stages. To overcome bottlenecks, the system designer faces the task of creating a design that performs well, yet is simple and modular.

This chapter describes techniques to avoid or hide performance bottlenecks. [Section 6.1](#) presents ways to identify bottlenecks and the general approaches to handle them, including exploiting workload properties, concurrent execution of operations, speculation, and batching. [Section 6.2](#) examines specific versions of the general techniques to attack the common problem of implementing multilevel memory systems efficiently. [Section 6.3](#) presents scheduling algorithms for services to choose which request to process first, if there are several waiting for service.

---

## 6.1 DESIGNING FOR PERFORMANCE

Performance bottlenecks show up in computer systems for two reasons. First, limits imposed by physics, technology, or economics restrict the rate of improvement in some dimensions of technology, while other dimensions improve rapidly. An obvious class of limits are the physical ones. The speed of light limits how fast signals travel from one end of a chip to the other, how many memory elements can be within a given latency from the processor, and how fast a network message can travel in the Internet. Many other physical limits appear in computer systems, such as power and heat dissipation.

These limits force a designer to make trade-offs. For example, by shrinking a chip, a designer can make the chip faster, but it also reduces the area from which heat can be dissipated. Worse, the power dissipation increases as the designer speeds up the chip. A related trade-off is between the speed of a laptop and its power consumption. A designer wants to minimize a laptop's power consumption so that the battery lasts longer, yet customers want laptops with fast processors and large, bright screens.

Physical limits are only a subset of the limits a designer faces; there are also algorithmic, reliability, and economic limits. More limits mean more trade-offs and a higher risk of bottlenecks.

The second reason bottlenecks surface in computer systems is that several clients may share a device. If a device is busy serving one client, other clients must wait until the device becomes available. This property forces the system designer to answer questions such as which client should receive the device first. Should the device first perform the request that requires little work, perhaps at the cost of delaying the request that requires a lot of work? The designer would like to devise a scheduling plan that doesn't starve some clients in favor of others, provides low turnaround time

for each individual client request, and has little overhead so that it can serve many clients. As we will see, it is impossible to maximize all of these goals simultaneously, and thus a designer must make trade-offs. Trade-offs may favor one class of requests over another and may result in bottlenecks for the unfavored classes of requests.

Designing for performance creates two major challenges in computer systems. First, one must consider the benefits of optimization in the context of technology improvements. Some bottlenecks are intrinsic ones; they require careful thinking to ensure that the system runs faster than the performance of the slowest stage. Some bottlenecks are technology dependent; time may eliminate these, as technology improves. Unfortunately, it is sometimes difficult to decide whether or not a bottleneck is intrinsic. Not uncommonly, a performance optimization for the next product release is irrelevant by the time the product ships because technology improvements have removed the bottleneck completely. This phenomenon is so common in computer design that it has led to formulation of the design hint: *when in doubt use brute force*. Sidebar 6.1 discusses this hint.

**Sidebar 6.1 Design Hint: When in Doubt use Brute Force** This chapter describes a few design hints that help a designer resolve trade-offs in the face of limits. These design hints are hints because they often guide the designer in the right direction, but sometimes they don't. In this book we cover only a few, but the interested reader should digest *Hints for computer system design* by B. Lampson, which presents many more practical guidelines in the form of hints [Suggestions for Further Reading 1.5.4].

The design hint “when in doubt use brute force” is a direct corollary of the  $d(\text{technology})/dt$  curve (see Section 1.4). Given computing technology's historical rate of improvement, it is typically wiser to choose simple algorithms that are well understood rather than complex, badly characterized algorithms. By the time the complex algorithm is fully understood, implemented, and debugged, new hardware might be able to execute the simple algorithm fast enough. Thompson and Ritchie used a fixed-size table of processes in the UNIX system and searched the table linearly because a table was simple to implement and the number of processes was small. With Joe Condon, Thompson also built the Belle chess machine that relied mostly on special-purpose hardware to search many positions per second rather than on sophisticated algorithms. Belle won the world computer chess championships several times in the late 1970s and early 1980s and achieved an ELO rating of 2250. (ELO is a numerical rating system used by the World Chess Federation (FIDE) to rank chess players; a rating of 2250 makes one a strong competitive player.) Later, as technology marched on, programs that performed brute-force searching algorithms on an off-the-shelf PC conquered the world computer chess championships. As of August 2005, the Hydra supercomputer (64 PCs, each with a chess coprocessor) is estimated by its creators to have an ELO rating of 3200, which is better than the best human player.

A second challenge in designing for performance is maintaining the simplicity of the design. For example, if the design uses different devices with approximately the same high-level function but radically different performance, a challenge is to abstract devices such that they can be used through a simple uniform interface. In this chapter, we see how a clever implementation of the READ and WRITE interface for memory can transparently extend the effective size of RAM to the size of a magnetic disk.

### 6.1.1 Performance Metrics

To understand bottlenecks more fully, recall that computer systems are organized in modules to achieve the benefits of modularity and that to process a request, the request may be handed from one module to another. For example, a camera may generate a continuous stream of requests containing video frames and send them to a service that digitizes each frame. The digitizing service in turn may send its output to a file service that stores the frames on a magnetic disk.

By describing this application in a client/service style, we can obtain some insights about important performance metrics. It is immediately clear that in a computer system such as this one, four metrics are of importance: the capacity of the service, its utilization, the time clients must wait for request to complete, and throughput, the rate at which services can handle requests. We will discuss each metric in turn.

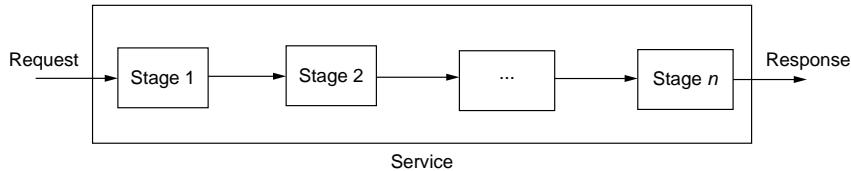
#### 6.1.1.1 Capacity, Utilization, Overhead, and Useful Work

Every service has some *capacity*, a consistent measure of a service's size or amount of resources. *Utilization* is the percentage of capacity of a resource that is used for some given workload of requests. A simple measure of processor capacity is cycles. For example, the processor might be utilized 10% for the duration of some workload, which means that 90% of its processor cycles are unused. For a magnetic disk, the capacity is usually measured in sectors. If a disk is utilized 80%, then 80% of its sectors are used to store data.

In a layered system, each layer may have a different view of the capacity and utilization of the underlying resources. For example, a processor may be 95% utilized but delivering only 70% of its cycles to the application because the operating system uses 25%. Each layer considers what the layers below it do to be *overhead* in time and space, and what the layers above it do to be *useful work*. In the processor example, from the application point of view, the 25% of cycles used by the operating system is overhead and the 70% is useful work. In the disk example, if 10% of the disk is used for storing file system data structures, then from the application point of view that 10% used by the file system is overhead and only 90% is useful capacity.

#### 6.1.1.2 Latency

*Latency* is the delay between a change at the input to a system and the corresponding change at its output. From the client/service perspective, the latency of a request is the time from issuing the request until the time the response is received from the service.

**FIGURE 6.1**

A simple service composed of several stages.

This latency has several components: the latency of sending a message to the service, the latency of processing the request, and the latency of sending a response back.

If a task, such as asking a service to perform a request, is a sequence of subtasks, we can think of the complete task as traversing stages of a pipeline, where each stage of the pipeline performs a subtask (see Figure 6.1). In our example, the first stage in the pipeline is sending the request, the second stage is the service digitizing the frame, the third stage is the file service storing the frame, and the final stage is sending a response back to the client.

With this pipeline model in mind, it is easy to see that latency of a pipeline with stages A and B is greater than or equal to the sum of the latencies for each stage in the pipeline:

$$\text{latency}_{A+B} \geq \text{latency}_A + \text{latency}_B$$

It is possibly greater because passing a request from one stage to another might add some latency. For example, if the stages correspond to different services, perhaps running on different computers connected by a network, then the overhead of passing requests from one stage to another may add enough latency that it cannot be ignored.

If the stages are of a single service, that additional latency is typically small (e.g., the overhead of invoking a procedure) and can usually be ignored for first-order analysis of performance. Thus, in this case, to predict the latency of a service that isn't running yet but is expected to perform two functions, A and B, with known latencies, a designer can approximate the joint latency of A and B by adding the latency of A and the latency of B.

#### 6.1.1.3 Throughput

*Throughput* is a measure of the rate of useful work done by a service for some given workload of requests. In the camera example, the throughput we might care about is how many frames per second the system can process because it may determine what quality camera we want to buy.

The throughput of a system with pipelined stages is less than or equal to the minimum of the throughput for each stage:

$$t\text{throughput}_{A+B} \leq \min(t\text{throughput}_A, t\text{throughput}_B)$$

Again, if the stages are of a single service, passing the request from one stage to another usually adds little overhead and has little impact on total throughput. Thus, for first-order analysis that overhead can be ignored, and the relation is usually close to equality.

Consider a computer system with two stages: one that is able to process data at a rate of 1,000 kilobytes per second and a second one at a rate of 100 kilobytes per second. If the fast stage generates one byte of output for each byte of input, the overall throughput must be less than or equal to 100 kilobytes per second. If there is negligible overhead in passing requests between the two stages, then the throughput of the system is equal to the throughput of the bottleneck stage, 100 kilobytes per second. In this case, the utilization of stage 1 is 10% and that of stage 2 is 100%.

When a stage processes requests serially, the throughput and the latency of a stage are directly related. The average number of requests a stage handles is inversely proportional to the average time to process a single request:

$$t\text{hroughput} = \frac{1}{latency}$$

If all stages process requests serially, the average throughput of the complete pipeline is inversely proportional to the average time a request spends in the pipeline. In these pipelines, reducing latency improves throughput, and the other way around.

When a stage processes requests concurrently, as we will see later in this chapter, there is *no* direct relationship between latency and throughput. For stages that process requests concurrently, an increase in throughput may *not* lead to a decrease in latency. A useful analogy is pipes through which water flows with a constant velocity. One can have several parallel pipes (or one fatter pipe), which improves throughput but doesn't change latency.

### 6.1.2 A Systems Approach to Designing for Performance

To gauge how much improvement we can hope for in reducing a bottleneck, we must identify and determine the performance of the slowest and the next-slowest bottleneck. To improve the throughput of a system in which all stages have equal throughput requires improving *all* stages. On the other hand, improving the stage that has a throughput that is 10 times lower than any other stage's throughput may result in a factor of 10 improvement in the throughput of the whole system. We might determine these bottlenecks by measurements or by using simple analytical calculations based on the performance characteristics of each bottleneck. In principle, the performance of any issue in a computer system can be explained, but sometimes it may require substantial digging to find the explanation; see, for example, the study by Perl and Sites on Windows NT's performance [Suggestions for Further Reading 6.4.1].

One should approach performance optimization from a systems point of view. This observation may sound trivial, but many person-years of work have disappeared in optimizing individual stages that resulted in small overall performance improvements. The reason that engineers are tempted to fine-tune a single stage is that

optimizations result in some measurable benefits. An individual engineer can design an optimization (e.g., replacing a slow algorithm with a faster algorithm, removing unnecessary expensive operations, reorganizing the code to have a fast path, etc.), implement it, and measure it, and can usually observe some performance improvement in that stage. This improvement stimulates the design of another optimization, which results in new benefits, and so on. Once one gets into this cycle, it is difficult to keep the *law of diminishing returns* in mind and realize that further improvements may result in little benefit to the system as a whole.

Since optimizing individual stages typically runs into the law of diminishing returns, an approach that focuses on overall performance is preferred. The iterative approach articulated in Section 1.5.2 achieves this goal because at each iteration the designer must consider whether or not the next iteration is worth performing. If the next iteration identifies a bottleneck that, if removed, shows diminished returns, the designer can stop. If the final performance is good enough, the designer's job is done. If the final performance doesn't meet the target, the designer may have to rethink the whole design or revisit the design specification.

The iterative approach for designing for performance has the following steps:

1. Measure the system to find out whether or not a performance enhancement is needed. If performance is a problem, identify which aspect of performance (throughput or latency) is the problem. For multistage pipelines in which stages process requests concurrently, there is no direct relationship between latency and throughput, so improving latency and improving throughput might require different techniques.
2. Measure again, this time to identify the performance bottleneck. The bottleneck may not be in the place the designer expected and may shift from one design iteration to another.
3. Predict the impact of the proposed performance enhancement with a simple back-of-the-envelope model. (We introduce a few simple models in this chapter.) This prediction includes determining where the next bottleneck will be. A quick way to determine the next bottleneck is to unrealistically assume that the planned performance enhancement will remove the current bottleneck and result in a stage with zero latency and infinite throughput. Under this assumption, determine the next bottleneck and calculate its performance. This calculation will result in one of two conclusions:
  - a. Removing the current bottleneck doesn't improve system performance significantly. In this case, stop iterating, and reconsider the whole design or revisit the requirements. Perhaps the designer can adjust the interfaces between stages with the goal of tolerating costly operations. We will discuss several approaches in the next sections.
  - b. Removing the current bottleneck is likely to improve the system performance. In this case, focus attention on the bottleneck stage. Consider brute-force methods of relieving the bottleneck stage (e.g., add more memory). Taking

advantage of the  $\frac{d(\text{technology})}{dt}$  curve may be less expensive than being clever. If brute-force methods won't relieve the bottleneck, be smart. For example, try to exploit properties of the workload or find better algorithms.

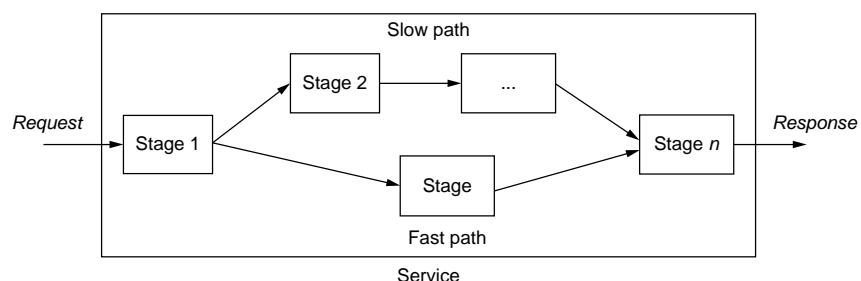
4. Measure the new implementation to verify that the change has the predicted impact. If not, revisit steps 1–3 and determine what went wrong.
5. Iterate. Repeat steps 1–5 until the performance meets the required level.

The rest of this chapter introduces various systems approaches to reducing latency and increasing throughput, as well as simple performance models to predict the resulting performance.

### 6.1.3 Reducing Latency by Exploiting Workload Properties

Reducing latency is difficult because the designer often runs into physical, algorithmic, and economic limits. For example, sending a message from a client on the east coast of the United States to a service on the west coast is dominated by the speed of light. Looking up an item in a hash table cannot go faster than the best algorithm for implementing hash tables. Building a very large memory that has uniform low latency is economically infeasible.

Once a designer has run into such limits, the common approach is to reduce the latency of some requests, perhaps even at the cost of increasing the latency for other requests. A designer may observe that certain requests are more common than other requests, and use that observation to improve the performance of the frequent operations by splitting the staged pipeline into a *fast path* for the frequent requests and a *slow path* for other requests (see Figure 6.2). For example, a service might remember the results of frequently asked requests so that when it receives a repeat of a recently handled request, it can return the remembered result immediately without having to recompute it. In practice, exploiting non-uniformity in applications



**FIGURE 6.2**

A simple service with a slow and fast path.

**Sidebar 6.2 Design Hint: Optimize for the Common Case** A cache (see Section 2.1.1.3) is the most common example of optimizing for the most frequent cases. We saw caches in the case study of the Domain Name System (in Section 4.4). As another example, consider a Web browser. Most Web browsers maintain a cache of recently accessed Web pages. This cache is indexed by the name of the Web page (e.g., <http://www.Scholarly.edu>) and returns the page for that name. If the user asks to view the same page again, then the cache can return the cached copy of the page immediately (a fast path); only the first access requires a trip to the service (a slow path). In addition to improving the user's interactive experience, the cache helps reduce the load on services and the load on the network. Because caches are so effective, many applications use several of them. For example, in addition to caching Web pages, many Web browsers have a cache to store the results of looking up names, such as “[www.Scholarly.edu](http://www.Scholarly.edu)”, so that the next request to “[www.Scholarly.edu](http://www.Scholarly.edu)” doesn't require a DNS lookup.

The design of multilevel memory in Section 6.2 is another example of how well a designer can exploit non-uniformity in a workload. Because applications have locality of reference, one can build large and fast memory systems out of a combination of a small but fast memory and a large but slow memory.

works so well that it has led to the design hint *optimize for the common case* (see Sidebar 6.2).

To evaluate the performance of systems with a fast and slow path, designers typically compute the average latency. If we know the latency of the fast and slow paths, and the frequency with which the system will take the fast path, then the average latency is:

$$\text{AverageLatency} = \text{Frequency}_{\text{fast}} \times \text{Latency}_{\text{fast}} + \text{Frequency}_{\text{slow}} \times \text{Latency}_{\text{slow}} \quad (6.1)$$

Whether introducing a fast path is worth the effort is dependent on the relative difference in latency between the fast and slow path, and the frequency with which the system can use the fast path, which is dependent on the workload. In addition, one might be able to change the design so that the fast path becomes faster at the cost of a slower slow path. If the frequency of taking the fast path is low, then introducing a fast path (and perhaps optimizing it at the cost of the slow path) is likely not worth the complexity. In practice, as we will see in Section 6.2, many workloads don't have a uniform distribution of requests, and introducing a fast path works well.

#### 6.1.4 Reducing Latency using Concurrency

Another way to reduce latency that may require some intellectual effort but that can be effective is to parallelize a stage. We take the processing that a stage must do for a single request and divide that processing up into subtasks that can be performed concurrently. Then, whenever several processors are available they can be assigned to run

those subtasks in parallel. The method can be applied either within a multiprocessor system or (if the subtasks aren't too entangled) with completely separate computers.

If the processing parallelizes perfectly (i.e., each subtask can run without any coordination with other subtasks and each subtask requires the same amount of work), then this plan can, in principle, speed up the processing by a factor  $n$ , where  $n$  is the number of subtasks executing in parallel. In practice, the speedup is usually less than  $n$  because there is overhead in parallelizing a computation—the subtasks need to communicate with each other, for example, to exchange intermediate results; because the subtasks do not require an equal amount of work; because the computation cannot be executed completely in parallel, so some fraction of the computation must be executed sequentially; or because the subtasks interfere with each other (e.g., they contend for a shared resource such as a lock, a shared memory, or a shared communication network).

Consider the processing that a search engine needs to perform in order to respond to a user search query. An early version of Google's search engine—described in more detail in Suggestions for Further Reading 3.2.4—parallelized this processing as follows. The search engine splits the index of the Web up in  $n$  pieces, each piece stored on a separate machine. When a front end receives a user query, it sends a copy of the query to each of the  $n$  machines. Each machine runs the query against its part of the index and sends the results back to the front end. The front end accumulates the results from the  $n$  machines, chooses a good order in which to display them, generates a Web page, and sends it to the user. This plan can give good speedup if the index is large and each of the  $n$  machines must perform a substantial, similar amount of computation. It is unlikely to achieve a full speedup of a factor  $n$  because there is parallelization overhead (to send the query to the  $n$  machines, receive  $n$  partial results, and merge them); because the amount of work is not balanced perfectly across the  $n$  machines and the front end must wait until the slowest responds; and because the work done by the front end in farming out the query and merging hasn't been parallelized.

Although parallelizing can improve performance, several challenges must be overcome. First, many applications are difficult to parallelize. Applications such as search have exploitable parallelism, but other computations don't split easily into  $n$  mostly independent pieces. Second, developing parallel applications is difficult because the programmer must manage the concurrency and coordinate the activities of the different subtasks. As we saw in Chapter 5, it is easy to get this wrong and introduce race conditions and deadlocks. Systems have been developed to make development of parallel applications easier, but they are often limited to a particular domain. The paper by Dean and Ghemawat [Suggestions for Further Reading 6.4.3] provides an example of how the programming and management effort can be minimized for certain stylized applications running in parallel on hundreds of machines. In general, however, programmers must often struggle with threads and locks, or explicit message passing, to obtain concurrency.

Because of these two challenges in parallelizing applications, designers traditionally have preferred to rely on continuous technology improvements to reduce application latency. However, physical and engineering limitations (primarily the problem of heat dissipation) are now leading processor manufacturers away from making processors

faster and toward placing several (and soon, probably, several hundred or even several thousand, as some are predicting [Suggestions for Further Reading 1.6.4]) processors on a single chip. This development means that improving performance by using concurrency will inevitably increase in importance.

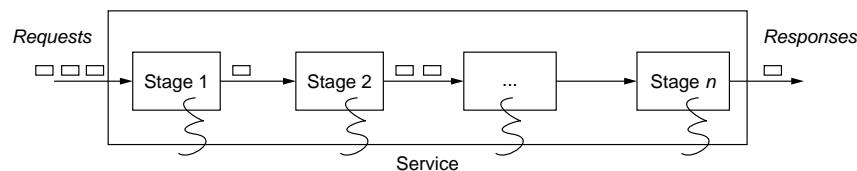
### 6.1.5 Improving Throughput: Concurrency

If the designer cannot reduce the latency of a request because of limits, an alternative approach is to *bide* the latency of a request by overlapping it with other requests. This approach doesn't improve the latency of an individual request, but it can improve system throughput. Because hiding latency is often much easier to achieve than improving latency, it has led to the hint: *instead of reducing latency, bide it* (see Sidebar 6.3). This section discusses how one can introduce concurrency in a multistage pipeline to increase throughput.

To overlap requests, we give each stage in the pipeline its own thread of computation so that it can compute concurrently, operating much like an assembly line (see Figure 6.3). If a stage has completed its task and has handed off the request to the next stage, then the stage can start processing the second request while the next stage processes the first request. In this fashion, the pipeline can work on several requests concurrently.

An implementation of this approach has two challenges. First, some stages of the pipeline may operate more slowly than other stages. As a result, one stage might not be able to hand off the request to the next stage because that next stage is still working on a previous request. As a result, a queue of requests may build up, while other stages might be idle. To ensure that a queue between two stages doesn't grow without bound, the stages are often coupled using a bounded buffer. We will discuss queuing in more detail in Section 6.1.6.

The second challenge is that several requests must be available. One natural source of multiple requests is if the system has several clients, each generating a request. A single client can also be a source of multiple requests if the client operates asynchronously. When an asynchronous client issues a request, rather than waiting for the response, it continues computing, perhaps issuing more requests. The main challenge



**FIGURE 6.3**

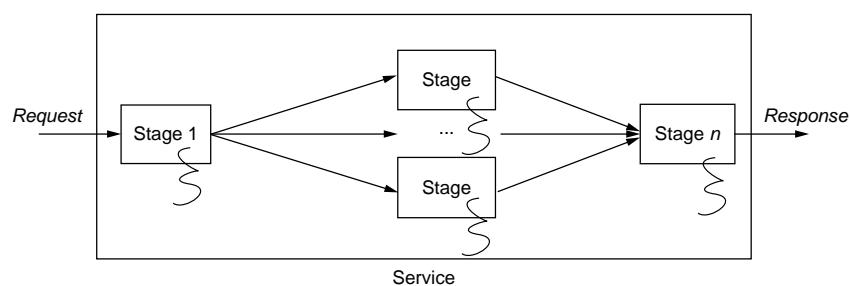
A simple service composed of several stages, with each stage operating concurrently using threads.

**Sidebar 6.3 Design Hint: Instead of Reducing Latency, Hide it** Latency is often not under the control of the designer but rather is imposed on the designer by physical properties such as the speed of light. Consider sending a message from the east coast of the United States to the west coast at the speed of light. This takes about 20 milliseconds (see Section 7.1 [online]); in the same time, a processor can execute millions of instructions. Worse, each new generation of processors gets faster every year, but the speed of light doesn't improve. As David Clark, a network researcher, put it succinctly: "One cannot bribe God." The speed of light shows up as an intrinsic barrier in many places of computer design, even when the distances are short. For example, dies are so large that for a signal to travel from one end of a chip to another is a bottleneck that limits the clock speed of a chip.

When a designer is faced with such intrinsic limits, the only option is to design systems that hide latency and try to exploit performance dimensions that do follow  $d(\text{technology})/dt$ . For example, transmission rates for data networks have improved dramatically, and so if a designer can organize the system such that communication can be overlapped with useful computation and many network requests can be batched into a large request, then the large request can be transferred efficiently. Many Web browsers use this strategy: while a large transfer runs in the background, users can continue browsing Web pages, hiding the latency of the transfer.

in issuing multiple requests asynchronously is that the client must then match the responses with the outstanding requests.

Once the system is organized to have many requests in flight concurrently, a designer may be able to improve throughput further by using *interleaving*. The idea is to make  $n$  instances of the bottleneck stage and run those  $n$  instances concurrently (see Figure 6.4). Stage 1 feeds the first request to instance 1, the second request to instance 2, and so on. If the throughput of a single instance is  $t$ , then the throughput using interleaving is  $n \times t$ , assuming enough requests are available to run all instances



**FIGURE 6.4**

Interleaving requests.

concurrently at full speed and the requests don't interfere with each other. The cost of interleaving is additional copies of the bottleneck stage.

RAID (see [Section 2.1.1.4](#)) interleaves several disks to achieve a high aggregate disk throughput. RAID 0 stripes the data across the disks: it stores block 0 on disk 0, block 1 on disk 1, and so on. If requests arrive for blocks on different disks, the RAID controller can serve those requests concurrently, improving throughput. In a similar style one can interleave memory chips to improve throughput. If the current instruction is stored in memory chip 0 and the next one is in memory chip 1, the processor can retrieve them concurrently. The cost of this design is the additional disks and memory chips, but often systems already have several memory chips or disks, in which case the added cost of interleaving can be small in comparison with the performance benefit.

### 6.1.6 Queuing and Overload

If a stage in [Figure 6.3](#) operates at its capacity (e.g., all physical processors are running threads), then a new request must wait until the stage becomes available; a queue of requests builds up waiting for the busy stage, while other stages may run idle. For example, the thread manager of [Section 5.5](#) maintains a table of threads, which records whether a thread is runnable; a runnable thread must wait until a processor is available to run it. The stage that runs with an input queue while other stages are running idle is a bottleneck.

Using queuing theory\* we can estimate the time that a request spends waiting in a queue for its turn to be processed (e.g., the time a thread spends in the ready queue). In queuing theory, the time that it takes to process a request (e.g., the time from when a thread starts running on the processor until it yields) is called the *service time*. The simplest queuing theory model assumes that requests (e.g., a thread entering the ready queue) arrive according to a random, memoryless process and have independent, exponentially distributed service times. In that case, a well-known queuing theory result tells us that the average queuing delay, measured in units of the average service time and including the service time of this request, will be  $1/(1-\rho)$ , where  $\rho$  is the service utilization. Thus, as the utilization approaches 1, the queuing delay will grow without bound.

This same phenomenon applies to the delays for threads waiting for a processor and to the delays that customers experience in supermarket checkout lines. Any time the demand for a service comes from many statistically independent sources, there will be fluctuations in the arrival of load and thus in the length of the queue at the bottleneck stage and the time spent waiting for service. The rate of arrival of requests for service is known as the *offered load*. Whenever the offered load is greater than the capacity of a service for some duration, the service is said to be *overloaded* for that time period.

---

\*The textbook by Jain is an excellent source to learn about queuing theory and how to reason about performance in computer systems [Suggestions for Further Reading 1.1.2].

In some constrained cases, where the designer can plan the system so that the capacity just matches the offered load of requests, it is possible to calculate the degree of concurrency necessary to achieve high throughput and the maximum length of the queue needed between stages. For example, suppose we have a processor that performs one instruction per nanosecond using a memory that takes 10 nanoseconds to respond. To avoid having the processor wait for the memory, it must make a memory request 10 instructions in advance of the instruction that needs it. If every instruction makes a request of memory, then by the time the memory responds, the processor will have issued 9 more. To avoid being a bottleneck, the memory therefore must be prepared to serve 10 requests concurrently.

If half of the instructions make a request of memory, then on average there will be five outstanding requests. Thus, a memory that can serve five requests concurrently would have enough capacity to keep up. To calculate the maximum length of the queue needed for this case depends on the application's pattern of memory references. For example, if every second instruction makes a memory request, a fixed-size queue of size five is sufficient to ensure that the queue never overflows. If the processor performs five instructions that make memory references followed by five that don't, then a fixed-size queue of size five will work, but the queue length will vary in length and the throughput will be different. If the requests arrive randomly, the queue can grow, in principle, without limit. If we were to use a memory that can handle 10 requests concurrently for this random pattern of memory references, then the memory would be utilized at 50% of capacity, and the average queue length would be  $(1/(1-0.5)) = 2$ . With this configuration, the processor observes latencies for some memory requests of 20 or more instruction cycles, and it is running much slower than the designer expected. This example illustrates that a designer must understand non-uniform patterns in the references to memory and exploit them to achieve good performance.

In many computer systems, the designer cannot plan the offered load that precisely, and thus stages will experience periods of overload. For example, an application may have several threads that become runnable all at the same time and there may not be enough processors available to run them. In such cases, at least occasional overload is inevitable. The significance of overload depends critically on how long it lasts. If the duration is comparable to the service time, then a queue is simply an orderly way to delay some requests for service until a later time when the offered load drops below the capacity of the service. Put another way, a queue handles short bursts of too much demand by time-averaging with adjacent periods when there is excess capacity.

If overload persists over long periods of time, the system designer has only two choices:

1. *Increase the capacity of the system.* If the system must meet the offered load, one approach is to design a system that has less overhead so that it can perform more useful work or purchase a better computer system with higher capacity. In computer systems, it is typically less expensive to buy the next generation of

the computer system that has higher capacity because of technology improvements than trying to squeeze the last ounce out of the implementation through complex algorithms.

2. *Shed load.* If purchasing a computer system with higher capacity isn't an option and system performance cannot be improved, the preferred method is to shed load by reducing or limiting the offered load until the load is less than the capacity of the system.

One approach to control the offered load is to use a bounded buffer (see [Figure 5.5](#)) between stages. When the bounded buffer ahead of the bottleneck stage is full, then the stage before it must wait until the bounded buffer empties a slot. Because the previous stage is waiting, its bounded buffer may fill up too, which may cause the stage before it to wait, and so on. The bottleneck may be pushed all the way back to the beginning of the pipeline. If this happens, the system cannot accept any more input, and what happens next depends on how the system is used.

If the source of the load needs the results of the output to generate the next request, then the load will be self-managing. This model of use applies to some interactive systems, in which the users cannot type the next command until the previous one finishes. This same idea will be used in [Chapter 7](#) [on-line] in the implementation of self-pacing network protocols.

If the source of the load decides not to make the request at all, then the offered load decreases. If the source, however, simply holds on to the request and resubmits it later, then the offered load doesn't decrease, but some requests are just deferred, perhaps to a time when the system isn't overloaded.

A crude approach to limiting a source is to put a *quota* on how many requests a source may have outstanding. For example, some systems enforce a rule that an application may not create more than some fixed number of active threads at the same time and may not have more than some fixed number of open files. If a source has reached its quota for a given service, the system denies the next request, limiting the offered load on the system.

An alternative to limiting the offered load is reducing it when a stage becomes overloaded. We will see one example of this approach in [Section 6.2](#). If the address spaces of a number of applications cannot fit in memory, the virtual memory manager can swap out a complete address space of one or more applications so that the remaining applications fit in memory. When the offered load decreases to normal levels, the virtual memory manager can swap in some of the applications that were swapped out.

### 6.1.7 Fighting Bottlenecks

If the designer cannot remove a bottleneck with the techniques described above, it may be possible instead to fight the bottleneck using one or more of three different techniques: batching, dallying, and speculation.

### 6.1.7.1 *Batching*

*Batching* is performing several requests as a group to avoid the setup overhead of doing them one at a time. Opportunities for batching arise naturally at a bottleneck stage, which may have a queue of requests waiting to be processed. For example, if a stage has several requests to send to the next stage, the stage can combine all of the messages into a single message and send that one message to the next stage. This use of batching divides the overhead of an expensive operation (e.g., sending a message) over the several messages. More generally, batching works well when processing a request has a fixed delay (e.g., transmitting the request) and a variable delay (e.g., performing the operation specified in the request). Without batching, processing  $n$  requests takes  $n \times (f + v)$ , where  $f$  is the fixed delay and  $v$  is the variable delay. With batching, processing  $n$  requests takes  $f + n \times v$ .

Once a stage performs batching, the potential arises for additional performance wins. Batching may create opportunities for the stage to avoid work. If two or more write requests in a batch are for the same disk block, then the stage can perform just the last one.

Batching may also provide opportunities to improve latency by *reordering* the processing of requests. As we will see in Section 6.3.4, if a disk controller receives a batch of requests, it can schedule them in an order that reduces the movement of the disk arm, reducing the total latency for the batch of requests.

### 6.1.7.2 *Dallying*

*Dallying* is delaying a request on the chance that the operation won't be needed, or to create more opportunities for batching. For example, a stage may delay a request that overwrites a disk block in the hope that a second one will come along for the same block. If a second one comes along, the stage can delete the first request and perform just the second one. As applied to writes, this benefit is sometimes called *write absorption*.

Dallying also increases the opportunities for batching. It purposely increases the latency of some requests in the hope that more requests will come along that can be combined with the delayed requests to form a batch. In this case, dallying increases the latency of some requests to improve the average latency of all requests.

A key design question in dallying is to decide how long to wait. There is no generic answer to this question. The costs and benefits of dallying are application and system specific.

### 6.1.7.3 *Speculation*

*Speculation* is performing an operation in advance of receiving a request on the chance that it will be requested. The goal is that the results can be delivered with less latency and perhaps with less setup overhead. Speculation can achieve this goal in two different ways. First, speculation can perform operations using otherwise idle resources. In this case, even if the speculation is wrong, performing the additional operations has no downside. Second, speculation can use a busy resource to do an

operation that has a long lead time so that the result of the operation can be available without waiting if it turns out to be needed. In this case, speculation might increase the delay and overhead of other requests without benefit because the prediction that the results may be needed might turn out to be wrong.

Speculating may sound bewildering because how can a computer system predict the input of an operation if it hasn't received the request yet, and how can it predict if the result of the operation will be useful in the future? Fortunately, many applications have request patterns that a system designer can exploit to predict an input. In some cases, the input value is evident; for example, a future instruction may add register 5 to register 9, and these register values may be available now. In some cases, the input values can be predicted accurately; for example, a program that asks to read byte  $n$  is likely to want to read bytes  $n + 1, n + 2$ , and so on, too. Similarly, for many applications a system can predict what results will be useful in the future. If a program performs instruction  $n$ , it will likely soon need the result of instruction  $n + 1$ ; only when the instruction  $n$  is a `JMP` will the prediction be wrong.

Sometimes a system can use speculation even if the system cannot predict accurately what the input to an operation is or whether the result will be useful. For example, if an input has only two values, then the system might create a new thread and have the main thread run with one input value and the second thread with the other input value. Later, when the system knows the value of the input, it terminates the thread that is computing with the wrong value and undoes any changes that thread might have made. This use of speculation becomes challenging when it involves shared state that is updated by different thread, but using techniques presented in [Chapter 9](#) [on-line] it is possible to undo the operations of a thread, even when shared state is involved.

Speculation creates more opportunities for batching and dallying. If the system speculates that a read request for block  $n$  will be followed by read requests for blocks  $n + 1$  through  $n + 8$ , then the system can batch those read requests. If a write request might soon be followed by another write request, the system can dally for a while to see if any others come in and, if so, batch all the writes together.

Key design questions associated with speculation are when to speculate and how much. Speculation can increase the load on later stages. If this increase in load results in a load higher than the capacity of a later stage, then requests must wait and latency will increase. Also, any work done that turns out to be not useful is overhead, and performing this unnecessary work may slow down other requests. There is no generic answer to this design question; instead, a designer must evaluate the benefits and cost of speculation in the context of the system.

#### **6.1.7.4 Challenges with Batching, Dallying, and Speculation**

Batching, dallying, and speculation introduce complexity because they introduce concurrency. The designer must coordinate incoming requests with the requests that are batched, dallied, or speculated. Furthermore, if the requested operations share variables, the designer must coordinate the references to these variables. Since coordination is difficult to get right, a designer must use these performance-enhancing

techniques with discipline. There is always the risk that by the time the designer has worked out the concurrency problems and the system has made it through the system tests, technology improvements will have made the extra complexity unnecessary. Problem set 14 explores several performance-enhancing techniques and their challenges with a simple multithreaded service.

### 6.1.8 An Example: The I/O Bottleneck

We illustrate design for performance using batching, dallying, and speculation through a case study involving a magnetic disk such as was described in [Sidebar 2.2](#). The performance problem with disks is that they are made of mechanical components. As a result, reading and writing data to a magnetic disk is slow compared to devices that have no mechanical components, such as RAM chips. The disk is therefore a bottleneck in many applications. This bottleneck is usually referred to as the *I/O bottleneck*.

Recall from [Sidebar 2.2](#) that the performance of reading and writing a disk block is determined by (1) the time to move the head to the appropriate track (the seek latency); (2) plus the time to wait until the requested sector rotates under the disk head (the rotational latency); (3) plus the time to transfer the data from the disk to the computer (the transfer latency).

The I/O bottleneck is getting worse over time. Seek latency and rotational latency are not improving as fast as processor performance. Thus, from the perspective of programs running on ever faster processors, I/O is getting slower over time. This problem is an example of problems due to incommensurate rates of technology improvement. Following the *incommensurate scaling rule* of [Chapter 1](#), applications and systems have been redesigned several times over the last few decades to cope with the I/O bottleneck.

To build some intuition for the I/O bottleneck, consider a typical disk of the last decade. The average seek latency (the time to move the head over one-third of the disk) is about 8 milliseconds. The disks spin at 7,200 rotations per minute, which is one rotation every 8.33 milliseconds. On average, the disk has to wait a half rotation for the desired block to be under the disk head; thus, the average rotational latency is 4.17 milliseconds.

Bits read from a disk encounter two potential transfer rate limits, either of which may become the bottleneck. The first limit is mechanical: the rate at which bits spin under the disk heads on their way to a buffer. The second limit is electrical: the rate at which the I/O channel or I/O bus can transfer the contents of the buffer to the computer. A typical modern 400-gigabyte disk has 16,383 cylinders, or about 24 megabytes per cylinder. That disk would probably have 8 two-sided platters and thus 16 read/write heads, so there would be  $24/16 = 1.5$  megabytes per track. When rotating at 7,200 revolutions per minute (120 revolutions per second), the bits will go by a head at  $120 \times 1.5 = 180$  megabytes per second. The I/O channel speed depends on which standard bus connects the disk to the computer. For the Integrated Device Electronics (IDE) bus, 66 megabytes per second is a common number in practice; for

## Chapter 4

# Concurrency Control

This chapter contains the book chapters:

R. Ramakrishnan and J. Gehrke. Database Management Systems. Third Edition. Chapters 16–17, pp. 519–544, and 549–575 (53 of 1065). McGraw-Hill, 2003. ISBN: 978-0-07-246563-1

While concurrency is a powerful technique for performance, it is unfortunately hard to get concurrent software right. The properties of **atomicity** and **durability**, and in particular the atomicity variant of *before-or-after atomicity* (also called *isolation*), are crucial for correctness, but particularly challenging to achieve when concurrency is employed. Gladly, a general theory of how to write correct, concurrent software, which respects before-or-after atomicity, over a memory abstraction has been developed in the context of database systems. This chapter discusses this theory, which is grounded on the notions of transactions, the ACID properties, and the serializability of transaction schedules. The chapter elaborates on multiple protocols for concurrency control, based on locking, timestamps, and multi-versioning. *The ultimate goal of this portion of the material is to enable us to design our own strategies for achieving atomicity when writing performance-optimized, concurrent services, and reflect on their correctness by equivalence to a well-known concurrency control protocol.*

The learning goals for this portion of the material are listed below.

- Identify the multiple interpretations of the property of atomicity.
- Implement methods to ensure before-or-after atomicity, and argue for their correctness.
- Explain the variants of the two-phase locking (2PL) protocol, in particular the widely-used Strict 2PL.
- Discuss definitions of serializability and their implications, in particular conflict-serializability and view-serializability.
- Apply the conflict-serializability test using a precedence graph to transaction schedules.
- Explain deadlock prevention and detection techniques.

- Apply deadlock detection using a waits-for graph to transaction schedules.
- Explain situations where predicate locking is required.
- Explain the optimistic concurrency control and multi-version concurrency control models.
- Predict validation decisions under optimistic concurrency control.



# 16

## OVERVIEW OF TRANSACTION MANAGEMENT

- What four properties of transactions does a DBMS guarantee?
- Why does a DBMS interleave transactions?
- What is the correctness criterion for interleaved execution?
- What kinds of anomalies can interleaving transactions cause?
- How does a DBMS use locks to ensure correct interleavings?
- What is the impact of locking on performance?
- What SQL commands allow programmers to select transaction characteristics and reduce locking overhead?
- How does a DBMS guarantee transaction atomicity and recovery from system crashes?
- **Key concepts:** ACID properties, atomicity, consistency, isolation, durability; schedules, serializability, recoverability, avoiding cascading aborts; anomalies, dirty reads, unrepeatable reads, lost updates; locking protocols, exclusive and shared locks, Strict Two-Phase Locking; locking performance, thrashing, hot spots; SQL transaction characteristics, savepoints, rollbacks, phantoms, access mode, isolation level; transaction manager, recovery manager, log, system crash, media failure; stealing frames, forcing pages; recovery phases, analysis, redo and undo.

I always say, keep a diary and someday it'll keep you.

—Mae West

In this chapter, we cover the concept of a *transaction*, which is the foundation for concurrent execution and recovery from system failure in a DBMS. A transaction is defined as *any one execution* of a user program in a DBMS and differs from an execution of a program outside the DBMS (e.g., a C program executing on Unix) in important ways. (Executing the same program several times generates several transactions.)

For performance reasons, a DBMS has to interleave the actions of several transactions. (We motivate interleaving of transactions in detail in Section 16.3.1.) However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect on the database) to some serial, or one-at-a-time, execution of the same set of transactions. How the DBMS handles concurrent executions is an important aspect of transaction management and the subject of *concurrency control*. A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion. The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of *crash recovery*. In this chapter, we provide a broad introduction to concurrency control and crash recovery in a DBMS. The details are developed further in the next two chapters.

In Section 16.1, we discuss four fundamental properties of database transactions and how the DBMS ensures these properties. In Section 16.2, we present an abstract way of describing an interleaved execution of several transactions, called a *schedule*. In Section 16.3, we discuss various problems that can arise due to interleaved execution. We introduce lock-based concurrency control, the most widely used approach, in Section 16.4. We discuss performance issues associated with lock-based concurrency control in Section 16.5. We consider locking and transaction properties in the context of SQL in Section 16.6. Finally, in Section 16.7, we present an overview of how a database system recovers from crashes and what steps are taken during normal execution to support crash recovery.

## 16.1 THE ACID PROPERTIES

We introduced the concept of database transactions in Section 1.7. To recapitulate briefly, a transaction is an execution of a user program, seen by the DBMS as a series of read and write operations.

A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as **atomic**: Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).
2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the **consistency** of the database. The DBMS assumes that consistency holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as **isolation**: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability**.

The acronym ACID is sometimes used to refer to these four properties of transactions: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

### 16.1.1 Consistency and Isolation

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that, when run to completion by itself against a ‘consistent’ database instance, the transaction will leave the database in a ‘consistent’ state. For example, the user may (naturally) have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts. To transfer money from one account to another, a transaction must debit one account, temporarily leaving the database inconsistent in a global sense, even though the new account balance may satisfy any integrity constraints with respect to the range of acceptable account balances. The user’s notion of a consistent database is preserved when the second account is credited with the transferred amount. If a faulty transfer program always credits the second account with one dollar less than the amount debited from the first account, the DBMS cannot be expected to detect inconsistencies due to such errors in the user program’s logic.

The isolation property is ensured by guaranteeing that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. (We discuss

how the DBMS implements this guarantee in Section 16.4.) For example, if two transactions  $T_1$  and  $T_2$  are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of)  $T_1$  followed by executing  $T_2$  or executing  $T_2$  followed by executing  $T_1$ . (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) results in a consistent final database instance.

**Database consistency** is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation, and transaction consistency. Next, we discuss how atomicity and durability are guaranteed in a DBMS.

### 16.1.2 Atomicity and Durability

Transactions can be incomplete for three kinds of reasons. First, a transaction can be **aborted**, or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew. Second, the system may crash (e.g., because the power supply is interrupted) while one or more transactions are in progress. Third, a transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).

Of course, since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Therefore, a DBMS must find a way to remove the effects of partial transactions from the database. That is, it must ensure transaction atomicity: Either all of a transaction's actions are carried out or none are. A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the *log*, of all writes to the database. The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

The DBMS component that ensures atomicity and durability, called the *recovery manager*, is discussed further in Section 16.7.

## 16.2 TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or *list*, of **actions**. The actions that can be executed by a transaction include **reads** and **writes** of *database objects*. To keep our notation simple, we assume that an object  $O$  is always read into a program variable that is also named  $O$ . We can therefore denote the action of a transaction  $T$  reading an object  $O$  as  $R_T(O)$ ; similarly, we can denote writing as  $W_T(O)$ . When the transaction  $T$  is clear from the context, we omit the subscript.

In addition to reading and writing, each transaction *must* specify as its final action either **commit** (i.e., complete successfully) or **abort** (i.e., terminate and undo all the actions carried out thus far).  $Abort_T$  denotes the action of  $T$  aborting, and  $Commit_T$  denotes  $T$  committing.

We make two important assumptions:

1. Transactions interact with each other *only* via database read and write operations; for example, they are not allowed to exchange messages.
2. A database is a *fixed* collection of *independent* objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

If the first assumption is violated, the DBMS has no way to detect or prevent inconsistencies caused by such external interactions between transactions, and it is up to the writer of the application to ensure that the program is well-behaved. We relax the second assumption in Section 16.6.2.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction  $T$  appear in a schedule must be the same as the order in which they appear in  $T$ . Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure 16.1 shows an execution order for actions of two transactions  $T_1$  and  $T_2$ . We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS*. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on; however, we assume that these actions do not affect other transactions; that is, the effect of a transaction on another transaction can be understood solely in terms of the common database objects that they read and write.

$T_1$	$T_2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure 16.1 A Schedule Involving Two Transactions

Note that the schedule in Figure 16.1 does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule**. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved—that is, transactions are executed from start to finish, one by one—we call the schedule a **serial schedule**.

## 16.3 CONCURRENT EXECUTION OF TRANSACTIONS

Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, but not all interleavings should be allowed. In this section, we consider what interleavings, or schedules, a DBMS should allow.

### 16.3.1 Motivation for Concurrent Execution

The schedule shown in Figure 16.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases **system throughput** (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in **response time**, or average time taken to complete a transaction.

### 16.3.2 Serializability

A **serializable schedule** over a set  $S$  of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over  $S$ . That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some* serial order.<sup>1</sup>

As an example, the schedule shown in Figure 16.2 is serializable. Even though the actions of  $T_1$  and  $T_2$  are interleaved, the result of this schedule is equivalent to running  $T_1$  (in its entirety) and then running  $T_2$ . Intuitively,  $T_1$ 's read and write of  $B$  is not influenced by  $T_2$ 's actions on  $A$ , and the net effect is the same if these actions are ‘swapped’ to obtain the serial schedule  $T_1; T_2$ .

$T_1$	$T_2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
$R(B)$	
$W(B)$	
	$R(B)$
	$W(B)$
	Commit
Commit	

Figure 16.2 A Serializable Schedule

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution. To see this, note that the two example transactions from Figure 16.2 can be interleaved as shown in Figure 16.3. This schedule, also serializable, is equivalent to the serial schedule  $T_2; T_1$ . If  $T_1$  and  $T_2$  are submitted concurrently to a DBMS, either of these schedules (among others) could be chosen.

The preceding definition of a serializable schedule does not cover the case of schedules containing aborted transactions. We extend the definition of serializable schedules to cover aborted transactions in Section 16.3.4.

---

<sup>1</sup>If a transaction prints a value to the screen, this ‘effect’ is not directly captured in the database. For simplicity, we assume that such values are also written into the database.

$T_1$	$T_2$
	$R(A)$
	$W(A)$
$R(A)$	
	$R(B)$
	$W(B)$
$W(A)$	
$R(B)$	
$W(B)$	
Commit	Commit

Figure 16.3 Another Serializable Schedule

Finally, we note that a DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution; that is, using a schedule that is not serializable. This can happen for two reasons. First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule (e.g., see Section 17.6.2). Second, SQL gives application programmers the ability to instruct the DBMS to choose non-serializable schedules (see Section 16.6).

### 16.3.3 Anomalies Due to Interleaved Execution

We now illustrate three main ways in which a schedule involving two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state. Two actions on the same data object **conflict** if at least one of them is a write. The three anomalous situations can be described in terms of when the actions of two transactions  $T_1$  and  $T_2$  conflict with each other: In a **write-read (WR) conflict**,  $T_2$  reads a data object previously written by  $T_1$ ; we define **read-write (RW)** and **write-write (WW)** conflicts similarly.

#### Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction  $T_2$  could read a database object  $A$  that has been modified by another transaction  $T_1$ , which has not yet committed. Such a read is called a **dirty read**. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions  $T_1$  and  $T_2$ , each of which, run alone, preserves database consistency:  $T_1$  transfers \$100 from  $A$  to  $B$ , and  $T_2$  increments both  $A$  and  $B$  by 6% (e.g., annual interest is deposited into these two accounts). Suppose

that the actions are interleaved so that (1) the account transfer program  $T_1$  deducts \$100 from account  $A$ , then (2) the interest deposit program  $T_2$  reads the current values of accounts  $A$  and  $B$  and adds 6% interest to each, and then (3) the account transfer program credits \$100 to account  $B$ . The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 16.4. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of  $A$  written by  $T_1$  is read by  $T_2$  before  $T_1$  has completed all its changes.

$T_1$	$T_2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Figure 16.4 Reading Uncommitted Data

The general problem illustrated here is that  $T_1$  may write some value into  $A$  that makes the database inconsistent. As long as  $T_1$  overwrites this value with a ‘correct’ value of  $A$  before committing, no harm is done if  $T_1$  and  $T_2$  run in some serial order, because  $T_2$  would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress. Such a requirement would be too restrictive: To transfer money from one account to another, a transaction *must* debit one account, temporarily leaving the database inconsistent, and then credit the second account, restoring consistency.

## Unrepeatable Reads (RW Conflicts)

The second way in which anomalous behavior could result is that a transaction  $T_2$  could change the value of an object  $A$  that has been read by a transaction  $T_1$ , while  $T_1$  is still in progress.

If  $T_1$  tries to read the value of  $A$  again, it will get a different result, even though it has not modified  $A$  in the meantime. This situation could not arise in a serial execution of two transactions; it is called an **unrepeatable read**.

To see why this can cause problems, consider the following example. Suppose that  $A$  is the number of available copies for a book. A transaction that places an order first reads  $A$ , checks that it is greater than 0, and then decrements it. Transaction  $T_1$  reads  $A$  and sees the value 1. Transaction  $T_2$  also reads  $A$  and sees the value 1, decrements  $A$  to 0 and commits. Transaction  $T_1$  then tries to decrement  $A$  and gets an error (if there is an integrity constraint that prevents  $A$  from becoming negative).

This situation can never arise in a serial execution of  $T_1$  and  $T_2$ ; the second transaction would read  $A$  and see 0 and therefore not proceed with the order (and so would not attempt to decrement  $A$ ).

## Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction  $T_2$  could overwrite the value of an object  $A$ , which has already been modified by a transaction  $T_1$ , while  $T_1$  is still in progress. Even if  $T_2$  does not read the value of  $A$  written by  $T_1$ , a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction  $T_1$  sets their salaries to \$2000 and transaction  $T_2$  sets their salaries to \$1000. If we execute these in the serial order  $T_1$  followed by  $T_2$ , both receive the salary \$1000; the serial order  $T_2$  followed by  $T_1$  gives each the salary \$2000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Note that neither transaction reads a salary value before writing it—such a write is called a **blind write**, for obvious reasons.

Now, consider the following interleaving of the actions of  $T_1$  and  $T_2$ :  $T_2$  sets Harry's salary to \$1000,  $T_1$  sets Larry's salary to \$2000,  $T_2$  sets Larry's salary to \$1000 and commits, and finally  $T_1$  sets Harry's salary to \$2000 and commits. The result is not identical to the result of either of the two possible serial

executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

The problem is that we have a **lost update**. The first transaction to commit,  $T_2$ , overwrote Larry's salary as set by  $T_1$ . In the serial order  $T_2$  followed by  $T_1$ , Larry's salary should reflect  $T_1$ 's update rather than  $T_2$ 's, but  $T_1$ 's update is 'lost'.

### 16.3.4 Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions.<sup>2</sup> Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A **serializable schedule** over a set  $S$  of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed* transactions in  $S$ .

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program  $T_1$  deducts \$100 from account  $A$ , then (2) an interest deposit program  $T_2$  reads the current values of accounts  $A$  and  $B$  and adds 6% interest to each, then commits, and then (3)  $T_1$  is aborted. The corresponding schedule is shown in Figure 16.5.

$T_1$	$T_2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Figure 16.5 An Unrecoverable Schedule

---

<sup>2</sup>We must also consider incomplete transactions for a rigorous discussion of system failures, because transactions that are active when the system fails are neither aborted nor committed. However, system recovery usually begins by aborting all active transactions, and for our informal discussion, considering schedules involving committed and aborted transactions is sufficient.

Now,  $T_2$  has read a value for  $A$  that should never have been there. (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If  $T_2$  had not yet committed, we could deal with the situation by *cascading* the abort of  $T_1$  and also aborting  $T_2$ ; this process recursively aborts any transaction that read data written by  $T_2$ , and so on. But  $T_2$  has already committed, and so we cannot undo its actions. We say that such a schedule is *unrecoverable*. In a **recoverable schedule**, transactions commit only after (and if!) all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to **avoid cascading aborts**.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction  $T_2$  overwrites the value of an object  $A$  that has been modified by a transaction  $T_1$ , while  $T_1$  is still in progress, and  $T_1$  subsequently aborts. All of  $T_1$ 's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before  $T_1$ 's changes. (We look at the details of how a transaction abort is handled in Chapter 18.) When  $T_1$  is aborted and its changes are undone in this manner,  $T_2$ 's changes are lost as well, even if  $T_2$  decides to commit. So, for example, if  $A$  originally had the value 5, then was changed by  $T_1$  to 6, and by  $T_2$  to 7, if  $T_1$  now aborts, the value of  $A$  becomes 5 again. Even if  $T_2$  commits, its change to  $A$  is inadvertently lost. A concurrency control technique called Strict 2PL, introduced in Section 16.4, can prevent this problem (as discussed in Section 17.1).

## 16.4 LOCK-BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol* to achieve this. A **lock** is a small bookkeeping object associated with a database object. A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. Different locking protocols use different types of locks, such as shared locks or exclusive locks, as we see next, when we discuss the Strict 2PL protocol.

### 16.4.1 Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called *Strict Two-Phase Locking*, or *Strict 2PL*, has two rules. The first rule is

1. If a transaction  $T$  wants to *read* (respectively, *modify*) an object, it first requests a **shared** (respectively, **exclusive**) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is

2. All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details. (We discuss how application programmers can select properties of transactions and control locking overhead in Section 16.6.3.)

In effect, the locking protocol allows only ‘safe’ interleavings of transactions. If two transactions access completely independent parts of the database, they concurrently obtain the locks they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one wants to modify it, their actions are effectively ordered serially—all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction  $T$  requesting a shared (respectively, exclusive) lock on object  $O$  as  $S_T(O)$  (respectively,  $X_T(O)$ ) and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 16.4. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance,  $T1$  could change  $A$  from 10 to 20, then  $T2$  (which reads the value 20 for  $A$ ) could change  $B$  from 100 to 200, and then  $T1$  would read the value 200 for  $B$ . If run serially, either  $T1$  or  $T2$  would execute first, and read the values 10 for  $A$  and 100 for  $B$ : Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, such interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as

before,  $T_1$  would obtain an exclusive lock on  $A$  first and then read and write  $A$  (Figure 16.6). Then,  $T_2$  would request a lock on  $A$ . However, this request

$T_1$	$T_2$
$X(A)$	
$R(A)$	
$W(A)$	

Figure 16.6 Schedule Illustrating Strict 2PL

cannot be granted until  $T_1$  releases its exclusive lock on  $A$ , and the DBMS therefore suspends  $T_2$ .  $T_1$  now proceeds to obtain an exclusive lock on  $B$ , reads and writes  $B$ , then finally commits, at which time its locks are released.  $T_2$ 's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure 16.7.

$T_1$	$T_2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

Figure 16.7 Schedule Illustrating Strict 2PL with Serial Execution

In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 16.8, which is permitted by the Strict 2PL protocol.

It can be shown that the Strict 2PL algorithm allows only serializable schedules. None of the anomalies discussed in Section 16.3.3 can arise if the DBMS implements Strict 2PL.

$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Figure 16.8 Schedule Following Strict 2PL with Interleaved Actions

### 16.4.2 Deadlocks

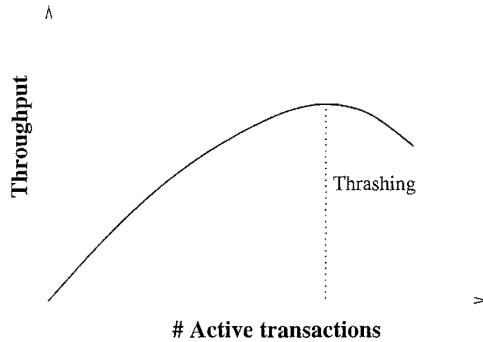
Consider the following example. Transaction  $T1$  sets an exclusive lock on object  $A$ ,  $T2$  sets an exclusive lock on  $B$ ,  $T1$  requests an exclusive lock on  $B$  and is queued, and  $T2$  requests an exclusive lock on  $A$  and is queued. Now,  $T1$  is waiting for  $T2$  to release its lock and  $T2$  is waiting for  $T1$  to release its lock. Such a cycle of transactions waiting for locks to be released is called a **deadlock**. Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations; the common approach is to detect and resolve deadlocks.

A simple way to identify deadlocks is to use a timeout mechanism. If a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it. We discuss deadlocks in more detail in Section 17.2.

## 16.5 PERFORMANCE OF LOCKING

Lock-based schemes are designed to resolve conflicts between transactions and use two basic mechanisms: *blocking* and *aborting*. Both mechanisms involve a performance penalty: Blocked transactions may hold locks that force other transactions to wait, and aborting and restarting a transaction obviously wastes the work done thus far by that transaction. A deadlock represents an extreme instance of blocking in which a set of transactions is forever blocked unless one of the deadlocked transactions is aborted by the DBMS.

In practice, fewer than 1% of transactions are involved in a deadlock, and there are relatively few aborts. Therefore, the overhead of locking comes primarily from delays due to blocking.<sup>3</sup> Consider how blocking delays affect throughput. The first few transactions are unlikely to conflict, and throughput rises in proportion to the number of active transactions. As more and more transactions execute concurrently on the same number of database objects, the likelihood of their blocking each other goes up. Thus, delays due to blocking increase with the number of active transactions, and throughput increases more slowly than the number of active transactions. In fact, there comes a point when adding another active transaction actually reduces throughput; the new transaction is blocked and effectively competes with (and blocks) existing transactions. We say that the system **thrashes** at this point, which is illustrated in Figure 16.9.



**Figure 16.9** Lock Thrashing

If a database system begins to thrash, the database administrator should reduce the number of transactions allowed to run concurrently. Empirically, thrashing is seen to occur when 30% of active transactions are blocked, and a DBA should monitor the fraction of blocked transactions to see if the system is at risk of thrashing.

Throughput can be increased in three ways (other than buying a faster system):

- By locking the smallest sized objects possible (reducing the likelihood that two transactions need the same lock).
- By reducing the time that transaction hold locks (so that other transactions are blocked for a shorter time).

---

<sup>3</sup>Many common deadlocks can be avoided using a technique called *lock downgrades*, implemented in most commercial systems (Section 17.3).

- By reducing **hot spots**. A hot spot is a database object that is frequently accessed and modified, and causes a lot of blocking delays. Hot spots can significantly affect performance.

The granularity of locking is largely determined by the database system's implementation of locking, and application programmers and the DBA have little control over it. We discuss how to improve performance by minimizing the duration locks are held and using techniques to deal with hot spots in Section 20.10.

## 16.6 TRANSACTION SUPPORT IN SQL

We have thus far studied transactions and transaction management using an abstract model of a transaction as a sequence of read, write, and abort/commit actions. We now consider what support SQL provides for users to specify transaction-level behavior.

### 16.6.1 Creating and Terminating Transactions

A transaction is automatically started when a user executes a statement that accesses either the database or the catalogs, such as a `SELECT` query, an `UPDATE` command, or a `CREATE TABLE` statement.<sup>4</sup>

Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a `COMMIT` command or a `ROLLBACK` (the SQL keyword for abort) command.

In SQL:1999, two new features are provided to support applications that involve long-running transactions, or that must run several transactions one after the other. To understand these extensions, recall that all the actions of a given transaction are executed in order, regardless of how the actions of different transactions are interleaved. We can think of each transaction as a sequence of steps.

The first feature, called a **savepoint**, allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results. This can be accomplished by defining savepoints.

---

<sup>4</sup>Some SQL statements—e.g., the `CONNECT` statement, which connects an application program to a database server—do not require the creation of a transaction.

**SQL:1999 Nested Transactions:** The concept of a transaction as an atomic sequence of actions has been extended in SQL:1999 through the introduction of the *savepoint* feature. This allows parts of a transaction to be selectively rolled back. The introduction of savepoints represents the first SQL support for the concept of **nested transactions**, which have been extensively studied in the research community. The idea is that a transaction can have several nested subtransactions, each of which can be selectively rolled back. Savepoints support a simple form of one-level nesting.

In a long-running transaction, we may want to define a series of savepoints. The savepoint command allows us to give each savepoint a name:

```
SAVEPOINT <savepoint name>
```

A subsequent rollback command can specify the savepoint to roll back to

```
ROLLBACK TO SAVEPOINT <savepoint name>
```

If we define three savepoints *A*, *B*, and *C* in that order, and then rollback to *A*, all operations since *A* are undone, including the creation of savepoints *B* and *C*. Indeed, the savepoint *A* is itself undone when we roll back to it, and we must re-establish it (through another savepoint command) if we wish to be able to roll back to it again. From a locking standpoint, locks obtained after savepoint *A* can be released when we roll back to *A*.

It is instructive to compare the use of savepoints with the alternative of executing a series of transactions (i.e., treat all operations in between two consecutive savepoints as a new transaction). The savepoint mechanism offers two advantages. First, we can roll back over several savepoints. In the alternative approach, we can roll back only the most recent transaction, which is equivalent to rolling back to the most recent savepoint. Second, the overhead of initiating several transactions is avoided.

Even with the use of savepoints, certain applications might require us to run several transactions one after the other. To minimize the overhead in such situations, SQL:1999 introduces another feature, called **chained transactions**. We can commit or roll back a transaction and immediately initiate another transaction. This is done by using the optional keywords AND CHAIN in the COMMIT and ROLLBACK statements.

### 16.6.2 What Should We Lock?

Until now, we have discussed transactions and concurrency control in terms of an abstract model in which a database contains a fixed collection of objects, and each transaction is a series of read and write operations on individual objects. An important question to consider in the context of SQL is what the DBMS should treat as an *object* when setting locks for a given SQL statement (that is part of a transaction).

Consider the following query:

```
SELECT S.rating, MIN (S.age)
FROM   Sailors S
WHERE  S.rating = 8
```

Suppose that this query runs as part of transaction  $T_1$  and an SQL statement that modifies the age of a given sailor, say Joe, with  $rating=8$  runs as part of transaction  $T_2$ . What ‘objects’ should the DBMS lock when executing these transactions? Intuitively, we must detect a conflict between these transactions.

The DBMS could set a shared lock on the entire *Sailors* table for  $T_1$  and set an exclusive lock on *Sailors* for  $T_2$ , which would ensure that the two transactions are executed in a serializable manner. However, this approach yields low concurrency, and we can do better by locking smaller objects, reflecting what each transaction actually accesses. Thus, the DBMS could set a shared lock on every row with  $rating=8$  for transaction  $T_1$  and set an exclusive lock on just the row for the modified tuple for transaction  $T_2$ . Now, other read-only transactions that do not involve  $rating=8$  rows can proceed without waiting for  $T_1$  or  $T_2$ .

As this example illustrates, the DBMS can lock objects at different **granularities**: We can lock entire tables or set row-level locks. The latter approach is taken in current systems because it offers much better performance. In practice, while row-level locking is generally better, the choice of locking granularity is complicated. For example, a transaction that examines several rows and modifies those that satisfy some condition might be best served by setting shared locks on the entire table and setting exclusive locks on those rows it wants to modify. We discuss this issue further in Section 17.5.3.

A second point to note is that SQL statements conceptually access a collection of rows described by a *selection predicate*. In the preceding example, transaction  $T_1$  accesses all rows with  $rating=8$ . We suggested that this could be dealt with by setting shared locks on all rows in *Sailors* that had  $rating=8$ . Unfortunately, this is a little too simplistic. To see why, consider an SQL statement that inserts

a new sailor with  $rating=8$  and runs as transaction  $T3$ . (Observe that this example violates our assumption of a fixed number of objects in the database, but we must obviously deal with such situations in practice.)

Suppose that the DBMS sets shared locks on every existing *Sailors* row with  $rating=8$  for  $T1$ . This does not prevent transaction  $T3$  from creating a brand new row with  $rating=8$  and setting an exclusive lock on this row. If this new row has a smaller *age* value than existing rows,  $T1$  returns an answer that depends on when it executed relative to  $T2$ . However, our locking scheme imposes no relative order on these two transactions.

This phenomenon is called the **phantom** problem: A transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. To prevent phantoms, the DBMS must conceptually lock *all possible* rows with  $rating=8$  on behalf of  $T1$ . One way to do this is to lock the entire table, at the cost of low concurrency. It is possible to take advantage of indexes to do better, as we will see in Section 17.5.1, but in general preventing phantoms can have a significant impact on concurrency.

It may well be that the application invoking  $T1$  can accept the potential inaccuracy due to phantoms. If so, the approach of setting shared locks on existing tuples for  $T1$  is adequate, and offers better performance. SQL allows a programmer to make this choice—and other similar choices—explicitly, as we see next.

### 16.6.3 Transaction Characteristics in SQL

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The **diagnostics size** determines the number of error conditions that can be recorded; we will not discuss this feature further.

If the **access mode** is **READ ONLY**, the transaction is not allowed to modify the database. Thus, **INSERT**, **DELETE**, **UPDATE**, and **CREATE** commands cannot be executed. If we have to execute one of these commands, the access mode should be set to **READ WRITE**. For transactions with **READ ONLY** access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concur-

rency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure 16.10. In this context, *dirty read* and *unrepeatable read* are defined as usual.

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Figure 16.10 Transaction Isolation Levels in SQL-92

The highest degree of isolation from the effects of other transactions is achieved by setting the isolation level for a transaction  $T$  to SERIALIZABLE. This isolation level ensures that  $T$  reads only the changes made by committed transactions, that no value read or written by  $T$  is changed by any other transaction until  $T$  is complete, and that if  $T$  reads a set of values based on some search condition, this set is not changed by other transactions until  $T$  is complete (i.e.,  $T$  avoids the phantom phenomenon).

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 17.5.1) and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that  $T$  reads only the changes made by committed transactions and no value read or written by  $T$  is changed by any other transaction until  $T$  is complete. However,  $T$  could experience the phantom phenomenon; for example, while  $T$  examines all Sailors records with  $rating=1$ , another transaction might add a new such Sailors record, which is missed by  $T$ .

A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it does not do index locking; that is, it locks only individual objects, not sets of objects. We discuss index locking in detail in Section 17.5.1.

READ COMMITTED ensures that  $T$  reads only the changes made by committed transactions, and that no value written by  $T$  is changed by any other transaction until  $T$  is complete. However, a value read by  $T$  may well be modified by

another transaction while  $T$  is still in progress, and  $T$  is exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction  $T$  can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while  $T$  is in progress, and  $T$  is also vulnerable to the phantom problem.

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself—a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level or even the READ UNCOMMITTED level, because a few incorrect or missing values do not significantly affect the result if the number of sailors is large.

The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

## 16.7 INTRODUCTION TO CRASH RECOVERY

The **recovery manager** of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of

committed transactions survive **system crashes**, (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The **transaction manager** of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol.<sup>5</sup> For simplicity of exposition, we make the following assumption:

**Atomic Writes:** Writing a page to disk is an atomic action.

This implies that the system does not crash while a write is in progress and is unrealistic. In practice, disk writes do not have this property, and steps must be taken during restart after a crash (Section 18.6) to verify that the most recent write to a given page was completed successfully, and to deal with the consequences if not.

### 16.7.1 Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:

1. Can the changes made to an object  $O$  in the buffer pool by a transaction  $T$  be written to disk before  $T$  commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the frame containing  $O$ ; of course, this page must have been unpinned by  $T$ . If such writes are allowed, we say that a **steal** approach is used. (Informally, the second transaction ‘steals’ a frame from  $T$ .)
2. When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so, we say that a **force** approach is used.

From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steal, force approach. If a no-steal approach is used, we do not have to undo the changes of an aborted transaction (because these changes have not been written to disk), and if a force approach is used, we do

---

<sup>5</sup>A concurrency control technique that does not involve locking could be used instead, but we assume that locking is used.

not have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs. If a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times. With a no-force approach, on the other hand, the in-memory copy of the page would be successively modified and written to disk just once, reflecting the effects of all 20 updates, when the page is eventually replaced in the buffer pool (in accordance with the buffer manager's page replacement policy).

For these reasons, most systems use a steal, no-force approach. Thus, if a frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active (*steal*); in addition, pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits (*no-force*).

### 16.7.2 Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions to enable it to perform its task in the event of a failure. In particular, a log of all modifications to the database is saved on **stable storage**, which is guaranteed<sup>6</sup> to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.

As discussed earlier in Section 16.7, it is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change. (Recall that this is the Write-Ahead Log, or WAL, property.)

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates

---

<sup>6</sup>Nothing in life is really guaranteed except death and taxes. However, we can reduce the chance of log failure to be vanishingly small by taking steps such as duplexing the log and storing the copies in different secure locations.

**Tuning the Recovery Subsystem:** DBMS performance can be greatly affected by the overhead imposed by the recovery subsystem. A DBA can take several steps to tune this subsystem, such as correctly sizing the log and how it is managed on disk, controlling the rate at which buffer pages are forced to disk, choosing a good frequency for checkpointing, and so forth.

to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log and written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process (while making sure that any log entries that describe changes these pages are written to disk first, i.e., following the WAL protocol). A process called *checkpointing*, which saves information about active transactions and dirty buffer pool pages, also helps reduce the time taken to recover from a crash. Checkpoints are discussed in Section 18.5.

### 16.7.3 Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases. In the **Analysis** phase, it identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash. In the **Redo** phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash. Finally, in the **Undo** phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions. The ARIES algorithm is discussed further in Chapter 18.

### 16.7.4 Atomicity: Implementing Rollback

It is important to recognize that the recovery subsystem is also responsible for executing the ROLLBACK command, which aborts a single transaction. Indeed,

the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash. All log records for a given transaction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback.

## 16.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the ACID properties? Define *atomicity*, *consistency*, *isolation*, and *durability* and illustrate them through examples. (**Section 16.1**)
- Define the terms *transaction*, *schedule*, *complete schedule*, and *serial schedule*. (**Section 16.2**)
- Why does a DBMS interleave concurrent transactions? (**Section 16.3**)
- When do two actions on the same data object *conflict*? Define the anomalies that can be caused by conflicting actions (*dirty reads*, *unrepeatable reads*, *lost updates*). (**Section 16.3**)
- What is a *serializable schedule*? What is a *recoverable schedule*? What is a schedule that *avoids cascading aborts*? What is a *strict schedule*? (**Section 16.3**)
- What is a *locking protocol*? Describe the *Strict Two-Phase Locking (Strict 2PL)* protocol. What can you say about the schedules allowed by this protocol? (**Section 16.4**)
- What overheads are associated with lock-based concurrency control? Discuss *blocking* and *aborting* overheads specifically and explain which is more important in practice. (**Section 16.5**)
- What is thrashing? What should a DBA do if the system thrashes? (**Section 16.5**)
- How can throughput be increased? (**Section 16.5**)
- How are transactions created and terminated in SQL? What are savepoints? What are chained transactions? Explain why savepoints and chained transactions are useful. (**Section 16.6**)
- What are the considerations in determining the locking granularity when executing SQL statements? What is the phantom problem? What impact does it have on performance? (**Section 16.6.2**)



# 17

## CONCURRENCY CONTROL

- ☛ How does Strict 2PL ensure serializability and recoverability?
- ☛ How are locks implemented in a DBMS?
- ☛ What are lock conversions and why are they important?
- ☛ How does a DBMS resolve deadlocks?
- ☛ How do current systems deal with the phantom problem?
- ☛ Why are specialized locking techniques used on tree indexes?
- ☛ How does multiple-granularity locking work?
- ☛ What is Optimistic concurrency control?
- ☛ What is Timestamp-Based concurrency control?
- ☛ What is Multiversion concurrency control?
- ☛ **Key concepts:** Two-phase locking (2PL), serializability, recoverability, precedence graph, strict schedule, view equivalence, view serializable, lock manager, lock table, transaction table, latch, convoy, lock upgrade, deadlock, waits-for graph, conservative 2PL, index locking, predicate locking, multiple-granularity locking, lock escalation, SQL isolation level, phantom problem, optimistic concurrency control, Thomas Write Rule, recoverability

Pooh was sitting in his house one day, counting his pots of honey, when there came a knock on the door.  
“Fourteen,” said Pooh. “Come in. Fourteen. Or was it fifteen? Bother. That’s muddled me.”

“Hallo, Pooh,” said Rabbit. “Hallo, Rabbit. Fourteen, wasn’t it?”  
 “What was?” “My pots of honey what I was counting.”  
 “Fourteen, that’s right.”  
 “Are you sure?”  
 “No,” said Rabbit. “Does it matter?”

—A.A. Milne, *The House at Pooh Corner*

In this chapter, we look at concurrency control in more detail. We begin by looking at locking protocols and how they guarantee various important properties of schedules in Section 17.1. Section 17.2 is an introduction to how locking protocols are implemented in a DBMS. Section 17.3 discusses the issue of lock conversions, and Section 17.4 covers deadlock handling. Section 17.5 discusses three specialized locking protocols—for locking sets of objects identified by some predicate, for locking nodes in tree-structured indexes, and for locking collections of related objects. Section 17.6 examines some alternatives to the locking approach.

## 17.1 2PL, SERIALIZABILITY, AND RECOVERABILITY

In this section, we consider how locking protocols guarantee some important properties of schedules; namely, serializability and recoverability. Two schedules are said to be **conflict equivalent** if they involve the (same set of) actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way.

As we saw in Section 16.3.3, two actions conflict if they operate on the same data object and at least one of them is a write. The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of nonconflicting operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database. Indeed, because they order all pairs of conflicting operations in the same way, we can obtain one of them from the other by repeatedly swapping pairs of nonconflicting actions, that is, by swapping pairs of actions whose relative order does not alter the outcome.

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule. Every conflict serializable schedule is serializable, if we assume that the set of items in the database does not grow or shrink; that is, values can be modified but items are not added or deleted. We make this assumption for now and consider its consequences in Section 17.5.1. However, some serializable schedules are not conflict serializable, as illustrated in Figure 17.1. This schedule is equivalent to executing the transactions serially in the order  $T_1, T_2$ ,

$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$ Commit	
$W(A)$ Commit		
		$W(A)$ Commit

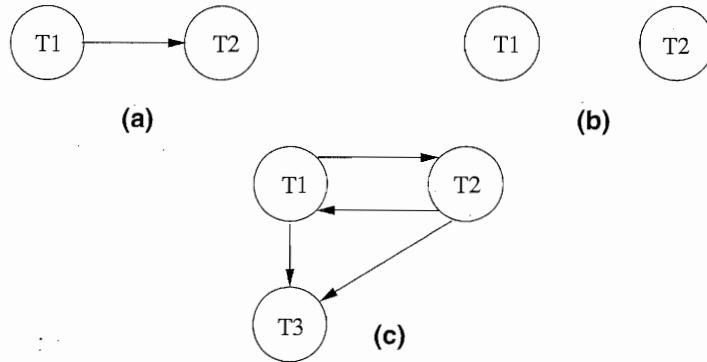
**Figure 17.1** Serializable Schedule That Is Not Conflict Serializable

$T3$ , but it is not conflict equivalent to this serial schedule because the writes of  $T1$  and  $T2$  are ordered differently.

It is useful to capture all potential conflicts between the transactions in a schedule in a **precedence graph**, also called a **serializability graph**. The precedence graph for a schedule  $S$  contains:

- A node for each committed transaction in  $S$ .
- An arc from  $T_i$  to  $T_j$  if an action of  $T_i$  precedes and conflicts with one of  $T_j$ 's actions.

The precedence graphs for the schedules shown in Figures 16.7, 16.8, and 17.1 are shown in Figure 17.2 (parts a, b, and c, respectively).

**Figure 17.2** Examples of Precedence Graphs

The Strict 2PL protocol (introduced in Section 16.4) allows only conflict serializable schedules, as is seen from the following two results:

1. A schedule  $S$  is conflict serializable if and only if its precedence graph is acyclic. (An equivalent serial schedule in this case is given by any topological sort over the precedence graph.)
2. Strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic.

A widely studied variant of Strict 2PL, called **Two-Phase Locking (2PL)**, relaxes the second rule of Strict 2PL to allow transactions to release locks before the end, that is, before the commit or abort action. For 2PL, the second rule is replaced by the following rule:

**(2PL) (2)** A transaction cannot request additional locks once it releases *any* lock.

Thus, every transaction has a ‘growing’ phase in which it acquires locks, followed by a ‘shrinking’ phase in which it releases locks.

It can be shown that even nonstrict 2PL ensures acyclicity of the precedence graph and therefore allows only conflict serializable schedules. Intuitively, an equivalent serial order of transactions is given by the order in which transactions enter their shrinking phase: If  $T_2$  reads or writes an object written by  $T_1$ ,  $T_1$  must have released its lock on the object before  $T_2$  requested a lock on this object. Thus,  $T_1$  precedes  $T_2$ . (A similar argument shows that  $T_1$  precedes  $T_2$  if  $T_2$  writes an object previously read by  $T_1$ . A formal proof of the claim would have to show that there is no cycle of transactions that ‘precede’ each other by this argument.)

A schedule is said to be **strict** if a value written by a transaction  $T$  is not read or overwritten by other transactions until  $T$  either aborts or commits. Strict schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified objects. (See the last example in Section 16.3.4.) Strict 2PL improves on 2PL by guaranteeing that every allowed schedule is strict in addition to being conflict serializable. The reason is that when a transaction  $T$  writes an object under Strict 2PL, it holds the (exclusive) lock until it commits or aborts. Thus, no other transaction can see or modify this object until  $T$  is complete.

The reader is invited to revisit the examples in Section 16.3.3 to see how the corresponding schedules are disallowed by Strict 2PL and 2PL. Similarly, it would be instructive to work out how the schedules for the examples in Section 16.3.4 are disallowed by Strict 2PL but not by 2PL.

### 17.1.1 View Serializability

Conflict serializability is sufficient but not necessary for serializability. A more general sufficient condition is view serializability. Two schedules  $S_1$  and  $S_2$  over the same set of transactions—any transaction that appears in either  $S_1$  or  $S_2$  must also appear in the other—are **view equivalent** under these conditions:

1. If  $T_i$  reads the initial value of object  $A$  in  $S_1$ , it must also read the initial value of  $A$  in  $S_2$ .
2. If  $T_i$  reads a value of  $A$  written by  $T_j$  in  $S_1$ , it must also read the value of  $A$  written by  $T_j$  in  $S_2$ .
3. For each data object  $A$ , the transaction (if any) that performs the final write on  $A$  in  $S_1$  must also perform the final write on  $A$  in  $S_2$ .

A schedule is **view serializable** if it is view equivalent to some serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 17.1 is view serializable, although it is not conflict serializable. Incidentally, note that this example contains blind writes. This is not a coincidence; it can be shown that any view serializable schedule that is not conflict serializable contains a blind write.

As we saw in Section 17.1, efficient locking protocols allow us to ensure that only conflict serializable schedules are allowed. Enforcing or testing view serializability turns out to be much more expensive, and the concept therefore has little practical use, although it increases our understanding of serializability.

## 17.2 INTRODUCTION TO LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the **lock manager**. The lock manager maintains a **lock table**, which is a hash table with the data object identifier as the key. The DBMS also maintains a descriptive entry for each transaction in a **transaction table**, and among other things, the entry contains a pointer to a list of locks held by the transaction. This list is checked before requesting a lock, to ensure that a transaction does not request the same lock twice.

A **lock table entry** for an object—which can be a page, a record, and so on, depending on the DBMS—contains the following information: the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

### 17.2.1 Implementing Lock and Unlock Requests

According to the Strict 2PL protocol, before a transaction  $T$  reads or writes a database object  $O$ , it must obtain a shared or exclusive lock on  $O$  and must hold on to the lock until it commits or aborts. When a transaction needs a lock on an object, it issues a lock request to the lock manager:

1. If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one).
2. If an exclusive lock is requested and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.
3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

When a transaction aborts or commits, it releases all its locks. When a lock on an object is released, the lock manager updates the lock table entry for the object and examines the lock request at the head of the queue for this object. If this request can now be granted, the transaction that made the request is woken up and given the lock. Indeed, if several requests for a shared lock on the object are at the front of the queue, all of these requests can now be granted together.

Note that if  $T_1$  has a shared lock on  $O$  and  $T_2$  requests an exclusive lock,  $T_2$ 's request is queued. Now, if  $T_3$  requests a shared lock, its request enters the queue behind that of  $T_2$ , even though the requested lock is compatible with the lock held by  $T_1$ . This rule ensures that  $T_2$  does not *starve*, that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent  $T_2$  from getting the exclusive lock for which it is waiting.

### Atomicity of Locking and Unlocking

The implementation of *lock* and *unlock* commands must ensure that these are atomic operations. To ensure atomicity of these operations when several instances of the lock manager code can execute concurrently, access to the lock table has to be guarded by an operating system synchronization mechanism such as a semaphore.

To understand why, suppose that a transaction requests an exclusive lock. The lock manager checks and finds that no other transaction holds a lock on the object and therefore decides to grant the request. But, in the meantime, another transaction might have requested and *received* a conflicting lock. To prevent this, the entire sequence of actions in a lock request call (checking to see if the request can be granted, updating the lock table, etc.) must be implemented as an atomic operation.

### Other Issues: Latches, Convoys

In addition to locks, which are held over a long duration, a DBMS also supports short-duration **latches**. Setting a latch before reading or writing a page ensures that the physical read or write operation is atomic; otherwise, two read/write operations might conflict if the objects being locked do not correspond to disk pages (the units of I/O). Latches are unset immediately after the physical read or write operation is completed.

We concentrated thus far on how the DBMS schedules transactions based on their requests for locks. This interleaving interacts with the operating system's scheduling of processes' access to the CPU and can lead to a situation called a **convoy**, where most of the CPU cycles are spent on process switching. The problem is that a transaction  $T$  holding a heavily used lock may be suspended by the operating system. Until  $T$  is resumed, every other transaction that needs this lock is queued. Such queues, called **convoys**, can quickly become very long; a convoy, once formed, tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preemptive scheduling.

## 17.3 LOCK CONVERSIONS

A transaction may need to acquire an exclusive lock on an object for which it already holds a shared lock. For example, a SQL update statement could result in shared locks being set on each row in a table. If a row satisfies the condition (in the WHERE clause) for being updated, an exclusive lock must be obtained for that row.

Such a **lock upgrade** request must be handled specially by granting the exclusive lock immediately if no other transaction holds a shared lock on the object and inserting the request at the front of the queue otherwise. The rationale for favoring the transaction thus is that it already holds a shared lock on the object and queuing it behind another transaction that wants an exclusive lock on the same object causes both a deadlock. Unfortunately, while favoring lock upgrades helps, it does not prevent deadlocks caused by two conflicting upgrade

requests. For example, if two transactions that hold a shared lock on an object both request an upgrade to an exclusive lock, this leads to a deadlock.

A better approach is to avoid the need for lock upgrades altogether by obtaining exclusive locks initially, and **downgrading** to a shared lock once it is clear that this is sufficient. In our example of an SQL update statement, rows in a table are locked in exclusive mode first. If a row does *not* satisfy the condition for being updated, the lock on the row is downgraded to a shared lock. Does the downgrade approach violate the 2PL requirement? On the surface, it does, because downgrading reduces the locking privileges held by a transaction, and the transaction may go on to acquire other locks. However, this is a special case, because the transaction did nothing but read the object that it downgraded, even though it conservatively obtained an exclusive lock. We can safely expand our definition of 2PL from Section 17.1 to allow lock downgrades in the growing phase, provided that the transaction has not modified the object.

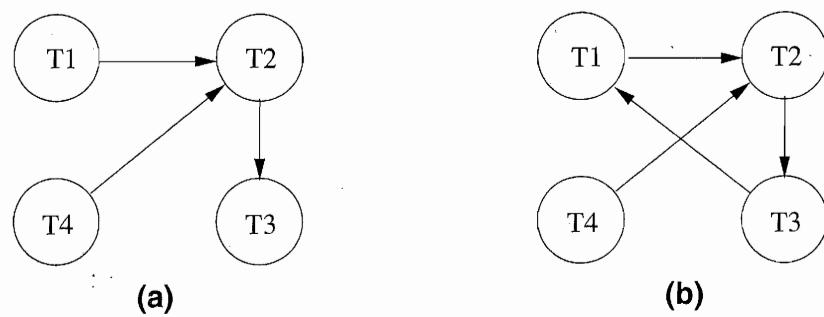
The downgrade approach reduces concurrency by obtaining write locks in some cases where they are not required. On the whole, however, it improves throughput by reducing deadlocks. This approach is therefore widely used in current commercial systems. Concurrency can be increased by introducing a new kind of lock, called an **update** lock, that is compatible with shared locks but not other update and exclusive locks. By setting an update lock initially, rather than exclusive locks, we prevent conflicts with other read operations. Once we are sure we need not update the object, we can downgrade to a shared lock. If we need to update the object, we must first upgrade to an exclusive lock. This upgrade does not lead to a deadlock because no other transaction can have an upgrade or exclusive lock on the object.

## 17.4 DEALING WITH DEADLOCKS

Deadlocks tend to be rare and typically involve very few transactions. In practice, therefore, database systems periodically check for deadlocks. When a transaction  $T_i$  is suspended because a lock that it requests cannot be granted, it must wait until all transactions  $T_j$  that currently hold conflicting locks release them. The lock manager maintains a structure called a **waits-for graph** to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from  $T_i$  to  $T_j$  if (and only if)  $T_i$  is waiting for  $T_j$  to release a lock. The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.

Consider the schedule shown in Figure 17.3. The last step, shown below the line, creates a cycle in the waits-for graph. Figure 17.4 shows the waits-for graph before and after this step.

$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	

**Figure 17.3** Schedule Illustrating Deadlock**Figure 17.4** Waits-for Graph Before and After Deadlock

Observe that the waits-for graph describes all active transactions, some of which eventually abort. If there is an edge from  $T_i$  to  $T_j$  in the waits-for graph, and both  $T_i$  and  $T_j$  eventually commit, there is an edge in the opposite direction (from  $T_j$  to  $T_i$ ) in the precedence graph (which involves only committed transactions).

The waits-for graph is periodically checked for cycles, which indicate deadlock. A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows some of the waiting transactions to proceed. The choice of which transaction to abort can be made using several criteria: the one with the fewest locks, the one that has done the least work, the one that is farthest from completion, and so on. Further, a transaction might have been repeatedly restarted; if so, it should eventually be favored during deadlock detection and allowed to complete.

A simple alternative to maintaining a waits-for graph is to identify deadlocks through a timeout mechanism: If a transaction has been waiting too long for a lock, we assume (pessimistically) that it is in a deadlock cycle and abort it.

#### 17.4.1 Deadlock Prevention

Empirical results indicate that deadlocks are relatively infrequent, and detection-based schemes work well in practice. However, if there is a high level of contention for locks and therefore an increased likelihood of deadlocks, prevention-based schemes could perform better. We can prevent deadlocks by giving each transaction a priority and ensuring that lower-priority transactions are not allowed to wait for higher-priority transactions (or vice versa). One way to assign priorities is to give each transaction a **timestamp** when it starts up. The lower the timestamp, the higher is the transaction's priority; that is, the oldest transaction has the highest priority.

If a transaction  $T_i$  requests a lock and transaction  $T_j$  holds a conflicting lock, the lock manager can use one of the following two policies:

- **Wait-die:** If  $T_i$  has higher priority, it is allowed to wait; otherwise, it is aborted.
- **Wound-wait:** If  $T_i$  has higher priority, abort  $T_j$ ; otherwise,  $T_i$  waits.

In the wait-die scheme, lower-priority transactions can never wait for higher-priority transactions. In the wound-wait scheme, higher-priority transactions never wait for lower-priority transactions. In either case, no deadlock cycle develops.

A subtle point is that we must also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that, in both schemes, the higher-priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and therefore the one with the highest priority, and will get all the locks it requires.

The wait-die scheme is nonpreemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs is never aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

A variant of 2PL, called **Conservative 2PL**, can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will be no deadlocks, and, perhaps more important, that a transaction that already holds some locks will not block waiting for other locks. If lock contention is heavy, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked. The trade-off is that a transaction acquires locks earlier, and if lock contention is low, locks are held longer under Conservative 2PL. From a practical perspective, it is hard to know exactly what locks are needed ahead of time, and this approach leads to setting more locks than necessary. It also has higher overhead for setting locks because a transaction has to release all locks and try to obtain them all over if it fails to obtain even one lock that it needs. This approach is therefore not used in practice.

## 17.5 SPECIALIZED LOCKING TECHNIQUES

Thus far we have treated a database as a *fixed* collection of *independent* data objects in our presentation of locking protocols. We now relax each of these restrictions and discuss the consequences.

If the collection of database objects is not fixed, but can grow and shrink through the insertion and deletion of objects, we must deal with a subtle complication known as the *phantom problem*, which was illustrated in Section 16.6.2. We discuss this problem in Section 17.5.1.

Although treating a database as an independent collection of objects is adequate for a discussion of serializability and recoverability, much better performance can sometimes be obtained using protocols that recognize and exploit the relationships between objects. We discuss two such cases, namely, locking in tree-structured indexes (Section 17.5.2) and locking a collection of objects with containment relationships between them (Section 17.5.3).

### 17.5.1 Dynamic Databases and the Phantom Problem

Consider the following example: Transaction  $T_1$  scans the *Sailors* relation to find the oldest sailor for each of the *rating* levels 1 and 2. First,  $T_1$  identifies and locks all pages (assuming that page-level locks are set) containing sailors with rating 1 and then finds the age of the oldest sailor, which is, say, 71. Next, transaction  $T_2$  inserts a new sailor with rating 1 and age 96. Observe that this new *Sailors* record can be inserted onto a page that does not contain other sailors with rating 1; thus, an exclusive lock on this page does not conflict with any of the locks held by  $T_1$ .  $T_2$  also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80).  $T_2$  then commits and releases its locks. Finally, transaction  $T_1$  identifies and locks pages containing (all remaining) sailors with rating 2 and finds the age of the oldest such sailor, which is, say, 63.

The result of the interleaved execution is that ages 71 and 63 are printed in response to the query. If  $T_1$  had run first, then  $T_2$ , we would have gotten the ages 71 and 80; if  $T_2$  had run first, then  $T_1$ , we would have gotten the ages 96 and 63. Thus, the result of the interleaved execution is not identical to any serial execution of  $T_1$  and  $T_2$ , even though both transactions follow Strict 2PL and commit. The problem is that  $T_1$  assumes that the pages it has locked include *all* pages containing *Sailors* records with rating 1, and this assumption is violated when  $T_2$  inserts a new such sailor on a different page.

The flaw is not in the Strict 2PL protocol. Rather, it is in  $T_1$ 's implicit assumption that it has locked the set of all *Sailors* records with *rating* value 1.  $T_1$ 's semantics requires it to identify all such records, but locking pages that contain such records *at a given time* does not prevent new “phantom” records from being added on other pages.  $T_1$  has therefore *not* locked the set of desired *Sailors* records.

Strict 2PL guarantees conflict serializability; indeed, there are no cycles in the precedence graph for this example because conflicts are defined with respect to objects (in this example, pages) read/written by the transactions. However, because the set of objects that *should* have been locked by  $T_1$  was altered by the actions of  $T_2$ , the outcome of the schedule differed from the outcome of any

serial execution. This example brings out an important point about conflict serializability: If new items are added to the database, conflict serializability does not guarantee serializability.

A closer look at how a transaction identifies pages containing Sailors records with *rating* 1 suggests how the problem can be handled:

- If there is no index and all pages in the file must be scanned,  $T_1$  must somehow ensure that no new pages are added to the file, in addition to locking all existing pages.
- If there is an index on the *rating* field,  $T_1$  can obtain a lock on the index page—again, assuming that physical locking is done at the page level—that contains a data entry with  $rating=1$ . If there are no such data entries, that is, no records with this *rating* value, the page that *would* contain a data entry for  $rating=1$  is locked to prevent such a record from being inserted. Any transaction that tries to insert a record with  $rating=1$  into the Sailors relation must insert a data entry pointing to the new record into this index page and is blocked until  $T_1$  releases its locks. This technique is called **index locking**.

Both techniques effectively give  $T_1$  a lock on the set of Sailors records with  $rating=1$ : Each existing record with  $rating=1$  is protected from changes by other transactions, and additionally, new records with  $rating=1$  cannot be inserted.

An independent issue is how transaction  $T_1$  can efficiently identify and lock the index page containing  $rating=1$ . We discuss this issue for the case of tree-structured indexes in Section 17.5.2.

We note that index locking is a special case of a more general concept called **predicate locking**. In our example, the lock on the index page implicitly locked all Sailors records that satisfy the logical predicate  $rating=1$ . More generally, we can support implicit locking of all records that match an arbitrary predicate. General predicate locking is expensive to implement and therefore not commonly used.

### 17.5.2 Concurrency Control in B+ Trees

A straightforward approach to concurrency control for B+ trees and ISAM indexes is to ignore the index structure, treat each page as a data object, and use some version of 2PL. This simplistic locking strategy would lead to very high lock contention in the higher levels of the tree, because every tree search begins at the root and proceeds along some path to a leaf node. Fortunately, much more efficient locking protocols that exploit the hierarchical structure of a tree

index are known to reduce the locking overhead while ensuring serializability and recoverability. We discuss some of these approaches briefly, concentrating on the search and insert operations.

Two observations provide the necessary insight:

1. The higher levels of the tree only direct searches. All the ‘real’ data is in the leaf levels (in the format of one of the three alternatives for data entries).
2. For inserts, a node must be locked (in exclusive mode, of course) only if a split can propagate up to it from the modified leaf.

Searches should obtain shared locks on nodes, starting at the root and proceeding along a path to the desired leaf. The first observation suggests that a lock on a node can be released as soon as a lock on a child node is obtained, because searches never go back up the tree.

A conservative locking strategy for inserts would be to obtain exclusive locks on all nodes as we go down from the root to the leaf node to be modified, because splits can propagate all the way from a leaf to the root. However, once we lock the child of a node, the lock on the node is required only in the event that a split propagates back to it. In particular, if the child of this node (on the path to the modified leaf) is not full when it is locked, any split that propagates up to the child can be resolved at the child, and does not propagate further to the current node. Therefore, when we lock a child node, we can release the lock on the parent if the child is not full. The locks held thus by an insert force any other transaction following the same path to wait at the earliest point (i.e., the node nearest the root) that might be affected by the insert. The technique of locking a child node and (if possible) releasing the lock on the parent is called **lock-coupling**, or **crabbing** (think of how a crab walks, and compare it to how we proceed down a tree, alternately releasing a lock on a parent and setting a lock on a child).

We illustrate B+ tree locking using the tree in Figure 17.5. To search for data entry 38\*, a transaction  $T_i$  must obtain an  $S$  lock on node  $A$ , read the contents and determine that it needs to examine node  $B$ , obtain an  $S$  lock on node  $B$  and release the lock on  $A$ , then obtain an  $S$  lock on node  $C$  and release the lock on  $B$ , then obtain an  $S$  lock on node  $D$  and release the lock on  $C$ .

$T_i$  always maintains a lock on one node in the path, to force new transactions that want to read or modify nodes on the same path to wait until the current transaction is done. If transaction  $T_j$  wants to delete 38\*, for example, it must also traverse the path from the root to node  $D$  and is forced to wait until  $T_i$

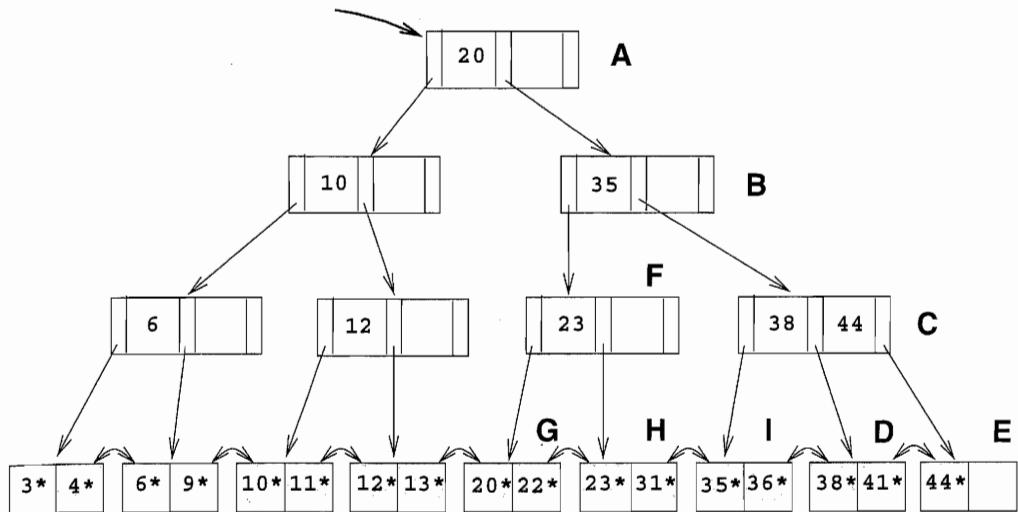


Figure 17.5 B+ Tree Locking Example

is done. Of course, if some transaction  $T_k$  holds a lock on, say, node  $C$  before  $T_i$  reaches this node,  $T_i$  is similarly forced to wait for  $T_k$  to complete.

To insert data entry  $45^*$ , a transaction must obtain an  $S$  lock on node  $A$ , obtain an  $S$  lock on node  $B$  and release the lock on  $A$ , then obtain an  $S$  lock on node  $C$  (observe that the lock on  $B$  is *not* released, because  $C$  is full), then obtain an  $X$  lock on node  $E$  and release the locks on  $C$  and then  $B$ . Because node  $E$  has space for the new entry, the insert is accomplished by modifying this node.

In contrast, consider the insertion of data entry  $25^*$ . Proceeding as for the insert of  $45^*$ , we obtain an  $X$  lock on node  $H$ . Unfortunately, this node is full and must be split. Splitting  $H$  requires that we also modify the parent, node  $F$ , but the transaction has only an  $S$  lock on  $F$ . Thus, it must request an upgrade of this lock to an  $X$  lock. If no other transaction holds an  $S$  lock on  $F$ , the upgrade is granted, and since  $F$  has space, the split does not propagate further and the insertion of  $25^*$  can proceed (by splitting  $H$  and locking  $G$  to modify the sibling pointer in  $I$  to point to the newly created node). However, if another transaction holds an  $S$  lock on node  $F$ , the first transaction is suspended until this transaction releases its  $S$  lock.

Observe that if another transaction holds an  $S$  lock on  $F$  and also wants to access node  $H$ , we have a deadlock because the first transaction has an  $X$  lock on  $H$ . The preceding example also illustrates an interesting point about sibling pointers: When we split leaf node  $H$ , the new node *must* be added to the *left* of  $H$ , since otherwise the node whose sibling pointer is to be changed would be node  $I$ , which has a different parent. To modify a sibling pointer on  $I$ , we

would have to lock its parent, node  $C$  (and possibly ancestors of  $C$ , in order to lock  $C$ ).

Except for the locks on intermediate nodes that we indicated could be released early, some variant of 2PL must be used to govern when locks can be released, to ensure serializability and recoverability.

This approach improves considerably on the naive use of 2PL, but several exclusive locks are still set unnecessarily and, although they are quickly released, affect performance substantially. One way to improve performance is for inserts to obtain shared locks instead of exclusive locks, except for the leaf, which is locked in exclusive mode. In the vast majority of cases, a split is not required and this approach works very well. If the leaf is full, however, we must upgrade from shared locks to exclusive locks for all nodes to which the split propagates. Note that such lock upgrade requests can also lead to deadlocks.

The tree locking ideas that we describe illustrate the potential for efficient locking protocols in this very important special case, but they are not the current state of the art. The interested reader should pursue the leads in the bibliography.

### 17.5.3 Multiple-Granularity Locking

Another specialized locking strategy, called **multiple-granularity locking**, allows us to efficiently set locks on objects that contain other objects.

For instance, a database contains several files, a file is a collection of pages, and a page is a collection of records. A transaction that expects to access most of the pages in a file should probably set a lock on the entire file, rather than locking individual pages (or records) when it needs them. Doing so reduces the locking overhead considerably. On the other hand, other transactions that require access to parts of the file—even parts not needed by this transaction—are blocked. If a transaction accesses relatively few pages of the file, it is better to lock only those pages. Similarly, if a transaction accesses several records on a page, it should lock the entire page, and if it accesses just a few records, it should lock just those records.

The question to be addressed is how a lock manager can efficiently ensure that a page, for example, is not locked by a transaction while another transaction holds a conflicting lock on the file containing the page (and therefore, implicitly, on the page).

The idea is to exploit the hierarchical nature of the ‘contains’ relationship. A database contains a set of files, each file contains a set of pages, and each page contains a set of records. This containment hierarchy can be thought of as a tree of objects, where each node contains all its children. (The approach can easily be extended to cover hierarchies that are not trees, but we do not discuss this extension.) A lock on a node locks that node and, implicitly, all its descendants. (Note that this interpretation of a lock is very different from B+ tree locking, where locking a node does *not* lock any descendants implicitly.)

In addition to shared (*S*) and exclusive (*X*) locks, multiple-granularity locking protocols also use two new kinds of locks, called **intention shared** (*IS*) and **intention exclusive** (*IX*) locks. *IS* locks conflict only with *X* locks. *IX* locks conflict with *S* and *X* locks. To lock a node in *S* (respectively, *X*) mode, a transaction must first lock all its ancestors in *IS* (respectively, *IX*) mode. Thus, if a transaction locks a node in *S* mode, no other transaction can have locked any ancestor in *X* mode; similarly, if a transaction locks a node in *X* mode, no other transaction can have locked any ancestor in *S* or *X* mode. This ensures that no other transaction holds a lock on an ancestor that conflicts with the requested *S* or *X* lock on the node.

A common situation is that a transaction needs to read an entire file and modify a few of the records in it; that is, it needs an *S* lock on the file and an *IX* lock so that it can subsequently lock some of the contained objects in *X* mode. It is useful to define a new kind of lock, called an *SIX* lock, that is logically equivalent to holding an *S* lock and an *IX* lock. A transaction can obtain a single *SIX* lock (which conflicts with any lock that conflicts with either *S* or *IX*) instead of an *S* lock and an *IX* lock.

A subtle point is that locks must be released in leaf-to-root order for this protocol to work correctly. To see this, consider what happens when a transaction  $T_i$  locks all nodes on a path from the root (corresponding to the entire database) to the node corresponding to some page  $p$  in *IS* mode, locks  $p$  in *S* mode, and then releases the lock on the root node. Another transaction  $T_j$  could now obtain an *X* lock on the root. This lock implicitly gives  $T_j$  an *X* lock on page  $p$ , which conflicts with the *S* lock currently held by  $T_i$ .

Multiple-granularity locking must be used with 2PL to ensure serializability. The 2PL protocol dictates when locks can be released. At that time, locks obtained using multiple-granularity locking can be released and must be released in leaf-to-root order.

Finally, there is the question of how to decide what granularity of locking is appropriate for a given transaction. One approach is to begin by obtaining fine granularity locks (e.g., at the record level) and, after the transaction requests

**Lock Granularity:** Some database systems allow programmers to override the default mechanism for choosing a lock granularity. For example, Microsoft SQL Server allows users to select page locking instead of table locking, using the keyword PAGLOCK. IBM's DB2 UDB allows for explicit table-level locking.

a certain number of locks at that granularity, to start obtaining locks at the next higher granularity (e.g., at the page level). This procedure is called **lock escalation**.

## 17.6 CONCURRENCY CONTROL WITHOUT LOCKING

Locking is the most widely used approach to concurrency control in a DBMS, but it is not the only one. We now consider some alternative approaches.

### 17.6.1 Optimistic Concurrency Control

Locking protocols take a pessimistic approach to conflicts between transactions and use either transaction abort or blocking to resolve conflicts. In a system with relatively light contention for data objects, the overhead of obtaining locks and following a locking protocol must nonetheless be paid.

In optimistic concurrency control, the basic premise is that most transactions do not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute. Transactions proceed in three phases:

1. **Read:** The transaction executes, reading values from the database and writing to a private workspace.
2. **Validation:** If the transaction decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, the transaction is aborted; its private workspace is cleared and it is restarted.
3. **Write:** If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.

If, indeed, there are few conflicts, and validation can be done efficiently, this approach should lead to better performance than locking. If there are many

conflicts, the cost of repeatedly restarting transactions (thereby wasting the work they've done) hurts performance significantly.

Each transaction  $T_i$  is assigned a timestamp  $TS(T_i)$  at the beginning of its validation phase, and the validation criterion checks whether the timestamp-ordering of transactions is an equivalent serial order. For every pair of transactions  $T_i$  and  $T_j$  such that  $TS(T_i) < TS(T_j)$ , one of the following **validation conditions** must hold:

1.  $T_i$  completes (all three phases) before  $T_j$  begins.
2.  $T_i$  completes before  $T_j$  starts its Write phase, and  $T_i$  does not write any database object read by  $T_j$ .
3.  $T_i$  completes its Read phase before  $T_j$  completes its Read phase, and  $T_i$  does not write any database object that is either read or written by  $T_j$ .

To validate  $T_j$ , we must check to see that one of these conditions holds with respect to each committed transaction  $T_i$  such that  $TS(T_i) < TS(T_j)$ . Each of these conditions ensures that  $T_j$ 's modifications are not visible to  $T_i$ .

Further, the first condition allows  $T_j$  to see some of  $T_i$ 's changes, but clearly, they execute completely in serial order with respect to each other. The second condition allows  $T_j$  to read objects while  $T_i$  is still modifying objects, but there is no conflict because  $T_j$  does not read any object modified by  $T_i$ . Although  $T_j$  might overwrite some objects written by  $T_i$ , all of  $T_i$ 's writes precede all of  $T_j$ 's writes. The third condition allows  $T_i$  and  $T_j$  to write objects at the same time and thus have even more overlap in time than the second condition, but the sets of objects written by the two transactions cannot overlap. Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met.

Checking these validation criteria requires us to maintain lists of objects read and written by each transaction. Further, while one transaction is being validated, no other transaction can be allowed to commit; otherwise, the validation of the first transaction might miss conflicts with respect to the newly committed transaction. The Write phase of a validated transaction must also be completed (so that its effects are visible outside its private workspace) before other transactions can be validated.

A synchronization mechanism such as a **critical section** can be used to ensure that at most one transaction is in its (combined) Validation/Write phases at any time. (When a process is executing a critical section in its code, the system suspends all other processes.) Obviously, it is important to keep these phases as short as possible in order to minimize the impact on concurrency. If copies of modified objects have to be copied from the private workspace, this

can make the Write phase long. An alternative approach (which carries the penalty of poor physical locality of objects, such as B+ tree leaf pages, that must be clustered) is to use a level of indirection. In this scheme, every object is accessed via a logical pointer, and in the Write phase, we simply switch the logical pointer to point to the version of the object in the private workspace, instead of copying the object.

Clearly, it is not the case that optimistic concurrency control has no overheads; rather, the locking overheads of lock-based approaches are replaced with the overheads of recording read-lists and write-lists for transactions, checking for conflicts, and copying changes from the private workspace. Similarly, the implicit cost of blocking in a lock-based approach is replaced by the implicit cost of the work wasted by restarted transactions.

## Improved Conflict Resolution<sup>1</sup>

Optimistic Concurrency Control using the three validation conditions described earlier is often overly conservative and unnecessarily aborts and restarts transactions. In particular, according to the validation conditions,  $T_i$  cannot write any object read by  $T_j$ . However, since the validation is aimed at ensuring that  $T_i$  logically executes before  $T_j$ , there is no harm if  $T_i$  writes all data items required by  $T_j$  before  $T_j$  reads them.

The problem arises because we have no way to tell when  $T_i$  wrote the object (relative to  $T_j$ 's reading it) at the time we validate  $T_j$ , since all we have is the list of objects written by  $T_i$  and the list read by  $T_j$ . Such false conflicts can be alleviated by a finer-grain resolution of data conflicts, using mechanisms very similar to locking.

The basic idea is that each transaction in the Read phase tells the DBMS about items it is reading, and when a transaction  $T_i$  is committed (and its writes are accepted), the DBMS checks whether any of the items written by  $T_i$  are being read by any (yet to be validated) transaction  $T_j$ . If so, we know that  $T_j$ 's validation must eventually fail. We can either allow  $T_j$  to discover this when it is validated (the **die** policy) or kill it and restart it immediately (the **kill** policy).

The details are as follows. Before reading a data item, a transaction  $T$  enters an **access entry** in a hash table. The access entry contains the *transaction id*, a *data object id*, and a *modified* flag (initially set to **false**), and entries are hashed on the data object id. A temporary exclusive lock is obtained on the

---

<sup>1</sup>We thank Alexander Thomasian for writing this section.

hash bucket containing the entry, and the lock is held while the read data item is copied from the database buffer into the private workspace of the transaction.

During validation of  $T$  the hash buckets of all data objects accessed by  $T$  are again locked (in exclusive mode) to check if  $T$  has encountered any data conflicts.  $T$  has encountered a conflict if the *modified* flag is set to **true** in one of its access entries. (This assumes that the ‘die’ policy is being used; if the ‘kill’ policy is used,  $T$  is restarted when the flag is set to **true**.)

If  $T$  is successfully validated, we lock the hash bucket of each object modified by  $T$ , retrieve all access entries for this object, set the *modified* flag to **true**, and release the lock on the bucket. If the ‘kill’ policy is used, the transactions that entered these access entries are restarted. We then complete  $T$ ’s Write phase.

It seems that the ‘kill’ policy is always better than the ‘die’ policy, because it reduces the overall response time and wasted processing. However, executing  $T$  to the end has the advantage that all of the data items required for its execution are prefetched into the database buffer, and restarted executions of  $T$  will not require disk I/O for reads. This assumes that the database buffer is large enough that prefetched pages are not replaced, and, more important, that **access invariance** prevails; that is, successive executions of  $T$  require the same data for execution. When  $T$  is restarted its execution time is much shorter than before because no disk I/O is required, and thus its chances of validation are higher. (Of course, if a transaction has already completed its Read phase once, subsequent conflicts should be handled using the ‘kill’ policy because all its data objects are already in the buffer pool.)

### 17.6.2 Timestamp-Based Concurrency Control

In lock-based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained, and the lock protocol extends this ordering on actions to transactions, thereby ensuring serializability. In optimistic concurrency control, a timestamp ordering is imposed on transactions and validation checks that all conflicting actions occurred in the same order.

Timestamps can also be used in another way: Each transaction can be assigned a timestamp at startup, and we can ensure, at execution time, that if action  $ai$  of transaction  $T_i$  conflicts with action  $aj$  of transaction  $T_j$ ,  $ai$  occurs before  $aj$  if  $TS(T_i) < TS(T_j)$ . If an action violates this ordering, the transaction is aborted and restarted.

To implement this concurrency control scheme, every database object  $O$  is given a **read timestamp**  $RTS(O)$  and a **write timestamp**  $WTS(O)$ . If transaction  $T$  wants to read object  $O$ , and  $TS(T) < WTS(O)$ , the order of this read with respect to the most recent write on  $O$  would violate the timestamp order between this transaction and the writer. Therefore,  $T$  is aborted and restarted with a new, larger timestamp. If  $TS(T) > WTS(O)$ ,  $T$  reads  $O$ , and  $RTS(O)$  is set to the larger of  $RTS(O)$  and  $TS(T)$ . (Note that a physical change—the change to  $RTS(O)$ —is written to disk and recorded in the log for recovery purposes, even on reads. This write operation is a significant overhead.)

Observe that if  $T$  is restarted with the same timestamp, it is guaranteed to be aborted again, due to the same conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention, where transactions are restarted with the *same* timestamp as before to avoid repeated restarts. This shows that the two uses of timestamps are quite different and should not be confused.

Next, consider what happens when transaction  $T$  wants to write object  $O$ :

1. If  $TS(T) < RTS(O)$ , the write action conflicts with the most recent read action of  $O$ , and  $T$  is therefore aborted and restarted.
2. If  $TS(T) < WTS(O)$ , a naive approach would be to abort  $T$  because its write action conflicts with the most recent write of  $O$  and is out of timestamp order. However, we can safely ignore such writes and continue. Ignoring outdated writes is called the **Thomas Write Rule**.
3. Otherwise,  $T$  writes  $O$  and  $WTS(O)$  is set to  $TS(T)$ .

## The Thomas Write Rule

We now consider the justification for the Thomas Write Rule. If  $TS(T) < WTS(O)$ , the current write action has, in effect, been made obsolete by the most recent write of  $O$ , which *follows* the current write according to the timestamp ordering. We can think of  $T$ 's write action as if it had occurred immediately *before* the most recent write of  $O$  and was never read by anyone.

If the Thomas Write Rule is not used, that is,  $T$  is aborted in case (2), the timestamp protocol, like 2PL, allows only conflict serializable schedules. If the Thomas Write Rule is used, some schedules are permitted that are not conflict serializable, as illustrated by the schedule in Figure 17.6.<sup>2</sup> Because  $T2$ 's write follows  $T1$ 's read and precedes  $T1$ 's write of the same object, this schedule is not conflict serializable.

---

<sup>2</sup>In the other direction, 2PL permits some schedules that are not allowed by the timestamp algorithm with the Thomas Write Rule; see Exercise 17.7.

$T1$	$T2$
$R(A)$	
	$W(A)$
$W(A)$	Commit
Commit	

**Figure 17.6** A Serializable Schedule That Is Not Conflict Serializable

The Thomas Write Rule relies on the observation that  $T2$ 's write is never seen by any transaction and the schedule in Figure 17.6 is therefore equivalent to the serializable schedule obtained by deleting this write action, which is shown in Figure 17.7.

$T1$	$T2$
$R(A)$	
	Commit
$W(A)$	
Commit	

**Figure 17.7** A Conflict Serializable Schedule

## Recoverability

Unfortunately, the timestamp protocol just presented permits schedules that are not recoverable, as illustrated by the schedule in Figure 17.8. If  $TS(T1) = 1$  and  $TS(T2) = 2$ , this schedule is permitted by the timestamp protocol (with or without the Thomas Write Rule). The timestamp protocol can be modified to disallow such schedules by **buffering** all write actions until the transaction commits. In the example, when  $T1$  wants to write  $A$ ,  $WTS(A)$  is updated to reflect this action, but the change to  $A$  is not carried out immediately; instead, it is recorded in a private workspace, or buffer. When  $T2$  wants to read  $A$  subsequently, its timestamp is compared with  $WTS(A)$ , and the read is seen to be permissible. However,  $T2$  is blocked until  $T1$  completes. If  $T1$  commits, its change to  $A$  is copied from the buffer; otherwise, the changes in the buffer are discarded.  $T2$  is then allowed to read  $A$ .

This blocking of  $T2$  is similar to the effect of  $T1$  obtaining an exclusive lock on  $A$ . Nonetheless, even with this modification, the timestamp protocol permits some schedules not permitted by 2PL; the two protocols are not quite the same. (See Exercise 17.7.)

$T1$	$T2$
$W(A)$	
	$R(A)$
	$W(B)$
	Commit

Figure 17.8 An Unrecoverable Schedule

Because recoverability is essential, such a modification must be used for the timestamp protocol to be practical. Given the added overhead this entails, on top of the (considerable) cost of maintaining read and write timestamps, timestamp concurrency control is unlikely to beat lock-based protocols in centralized systems. Indeed, it has been used mainly in the context of distributed database systems (Chapter 22).

### 17.6.3 Multiversion Concurrency Control

This protocol represents yet another way of using timestamps, assigned at startup time, to achieve serializability. The goal is to ensure that a transaction never has to wait to read a database object, and the idea is to maintain several versions of each database object, each with a write timestamp, and let transaction  $T_i$  read the most recent version whose timestamp precedes  $TS(T_i)$ .

If transaction  $T_i$  wants to write an object, we must ensure that the object has not already been read by some other transaction  $T_j$  such that  $TS(T_i) < TS(T_j)$ . If we allow  $T_i$  to write such an object, its change should be seen by  $T_j$  for serializability, but obviously  $T_j$ , which read the object at some time in the past, will not see  $T_i$ 's change.

To check this condition, every object also has an associated read timestamp, and whenever a transaction reads the object, the read timestamp is set to the maximum of the current read timestamp and the reader's timestamp. If  $T_i$  wants to write an object  $O$  and  $TS(T_i) < RTS(O)$ ,  $T_i$  is aborted and restarted with a new, larger timestamp. Otherwise,  $T_i$  creates a new version of  $O$  and sets the read and write timestamps of the new version to  $TS(T_i)$ .

The drawbacks of this scheme are similar to those of timestamp concurrency control, and in addition, there is the cost of maintaining versions. On the other hand, reads are never blocked, which can be important for workloads dominated by transactions that only read values from the database.

**What Do Real Systems Do?** IBM DB2, Informix, Microsoft SQL Server, and Sybase ASE use Strict 2PL or variants (if a transaction requests a lower than SERIALIZABLE SQL isolation level; see Section 16.6). Microsoft SQL Server also supports modification timestamps so that a transaction can run without setting locks and validate itself (do-it-yourself Optimistic Concurrency Control!). Oracle 8 uses a multiversion concurrency control scheme in which readers never wait; in fact, readers never get locks and detect conflicts by checking if a block changed since they read it. All these systems support multiple-granularity locking, with support for table, page, and row level locks. All deal with deadlocks using waits-for graphs. Sybase ASIQ supports only table-level locks and aborts a transaction if a lock request fails—updates (and therefore conflicts) are rare in a data warehouse, and this simple scheme suffices.

## 17.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- When are two schedules *conflict equivalent*? What is a *conflict serializable* schedule? What is a *strict* schedule? (**Section 17.1**)
- What is a *precedence graph* or *serializability graph*? How is it related to conflict serializability? How is it related to two-phase locking? (**Section 17.1**)
- What does the *lock manager* do? Describe the *lock table* and *transaction table* data structures and their role in lock management. (**Section 17.2**)
- Discuss the relative merits of *lock upgrades* and *lock downgrades*. (**Section 17.3**)
- Describe and compare deadlock detection and deadlock prevention schemes. Why are detection schemes more commonly used? (**Section 17.4**)
- If the collection of database objects is not fixed, but can grow and shrink through insertion and deletion of objects, we must deal with a subtle complication known as the *phantom problem*. Describe this problem and the index locking approach to solving the problem. (**Section 17.5.1**)
- In tree index structures, locking higher levels of the tree can become a performance bottleneck. Explain why. Describe specialized locking techniques that address the problem, and explain why they work correctly despite not being two-phase. (**Section 17.5.2**)
- *Multiple-granularity locking* enables us to set locks on objects that contain other objects, thus implicitly locking all contained objects. Why is this approach important and how does it work? (**Section 17.5.3**)

- In *optimistic concurrency control*, no locks are set and transactions read and modify data objects in a private workspace. How are conflicts between transactions detected and resolved in this approach? (**Section 17.6.1**)
- In *timestamp-based concurrency control*, transactions are assigned a timestamp at startup; how is it used to ensure serializability? How does the *Thomas Write Rule* improve concurrency? (**Section 17.6.2**)
- Explain why timestamp-based concurrency control allows schedules that are not recoverable. Describe how it can be modified through *buffering* to disallow such schedules. (**Section 17.6.2**)
- Describe *multiversion concurrency control*. What are its benefits and disadvantages in comparison to locking? (**Section 17.6.3**)

## EXERCISES

**Exercise 17.1** Answer the following questions:

1. Describe how a typical lock manager is implemented. Why must lock and unlock be atomic operations? What is the difference between a lock and a *latch*? What are *convoys* and how should a lock manager handle them?
2. Compare *lock downgrades* with upgrades. Explain why downgrades violate 2PL but are nonetheless acceptable. Discuss the use of *update locks* in conjunction with lock downgrades.
3. Contrast the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
4. State and justify the Thomas Write Rule.
5. Show that, if two schedules are conflict equivalent, then they are view equivalent.
6. Give an example of a serializable schedule that is not strict.
7. Give an example of a strict schedule that is not serializable.
8. Motivate and describe the use of locks for improved conflict resolution in Optimistic Concurrency Control.

**Exercise 17.2** Consider the following classes of schedules: *serializable*, *conflict-serializable*, *view-serializable*, *recoverable*, *avoids-cascading-aborts*, and *strict*. For each of the following schedules, state which of the preceding classes it belongs to. If you cannot decide whether a schedule belongs in a certain class based on the listed actions, explain briefly.

The actions are listed in the order they are scheduled and prefixed with the transaction name. If a commit or abort is not shown, the schedule is incomplete; assume that abort or commit must follow all the listed actions.

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)
2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)

3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)
4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)
5. T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
7. T1:W(X), T2:R(X), T1:W(X), T2:Abort, T1:Commit
8. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. T2: R(X), T3:W(X), T3:Commit, T1:W(Y), T1:Commit, T2:R(Y),  
T2:W(Z), T2:Commit
11. T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit

**Exercise 17.3** Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule, and Multiversion. For each of the schedules in Exercise 17.2, state which of these protocols allows it, that is, allows the actions to occur in exactly the order shown.

For the timestamp-based protocols, assume that the timestamp for transaction  $T_i$  is  $i$  and that a version of the protocol that ensures recoverability is used. Further, if the Thomas Write Rule is used, show the equivalent serial schedule.

**Exercise 17.4** Consider the following sequences of actions, listed in the order they are submitted to the DBMS:

- **Sequence S1:** T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y),  
T1:Commit, T2:Commit, T3:Commit
- **Sequence S2:** T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y),  
T1:Commit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control mechanisms, describe how the concurrency control mechanism handles the sequence.

Assume that the timestamp of transaction  $T_i$  is  $i$ . For lock-based concurrency control mechanisms, add lock and unlock requests to the previous sequence of actions as per the locking protocol. The DBMS processes actions in the order shown. If a transaction is blocked, assume that all its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

1. Strict 2PL with timestamps used for deadlock prevention.
2. Strict 2PL with deadlock detection. (Show the waits-for graph in case of deadlock.)
3. Conservative (and Strict, i.e., with locks held until end-of-transaction) 2PL.
4. Optimistic concurrency control.
5. Timestamp concurrency control with buffering of reads and writes (to ensure recoverability) and the Thomas Write Rule.
6. Multiversion concurrency control.



## Chapter 5

# Recovery

This chapter contains the book chapter:

R. Ramakrishnan and J. Gehrke. Database Management Systems. Third Edition. Chapter 18, pp. 579–600 (22 of 1065). McGraw-Hill, 2003. ISBN: 978-0-07-246563-1

**Atomicity** and **durability** are challenging in important ways other than with concurrency. First, we must deal with the risk of partial execution of functions in our service, e.g., due to errors, leading to data corruption and compromising *all-or-nothing atomicity*. Second, if volatile memory is employed, even complete execution of a function may not guarantee durability of its effects. Third, performance optimizations, such as use of two-level memories, may complicate the determination of the current state of the system in the event of a failure. Fortunately, methodologies to achieve atomicity and durability in the face of these challenges have been developed in the context of database systems. Log-based recovery techniques allow us to survive both fail-stop and media failures. These techniques have been heavily studied, and we will focus on a family of recovery algorithms called *ARIES* (*Algorithms for Recovery and Isolation Exploiting Semantics*). *The ultimate goal of this portion of the material is to provide us with a clear conceptual framework to reflect about recovery strategies, and to predict how different recovery strategies behave in different scenarios.*

The learning goals for this portion of the material are listed below.

- Explain the concepts of volatile, nonvolatile, and stable storage as well as the main assumptions underlying database recovery.
- Predict how force/no-force and steal/no-steal strategies for writes and buffer management influence the need for redo and undo.
- Explain the notion of logging and the concept of write-ahead logging.
- Predict what portions of the log and database are necessary for recovery under different failure scenarios.
- Explain how write-ahead logging is achieved in the ARIES protocol.
- Explain the functions of recovery metadata such as the transaction table and the dirty page table.

- Predict how recovery metadata is updated during normal operation.
- Interpret the contents of the log resulting from ARIES normal operation.
- Explain the three phases of ARIES crash recovery: analysis, redo, and undo.
- Predict how recovery metadata, system state, and the log are updated during recovery.



# 18

## CRASH RECOVERY

- ☛ What steps are taken in the ARIES method to recover from a DBMS crash?
- ☛ How is the log maintained during normal operation?
- ☛ How is the log used to recover from a crash?
- ☛ What information in addition to the log is used during recovery?
- ☛ What is a checkpoint and why is it used?
- ☛ What happens if repeated crashes occur during recovery?
- ☛ How is media failure handled?
- ☛ How does the recovery algorithm interact with concurrency control?
- ☛ **Key concepts:** steps in recovery, analysis, redo, undo; ARIES, repeating history; log, LSN, forcing pages, WAL; types of log records, update, commit, abort, end, compensation; transaction table, lastLSN; dirty page table, recLSN; checkpoint, fuzzy checkpointing, master log record; media recovery; interaction with concurrency control; shadow paging

Humpty Dumpty sat on a wall.  
Humpty Dumpty had a great fall.  
All the King's horses and all the King's men  
Could not put Humpty together again.

—Old nursery rhyme

The **recovery manager** of a DBMS is responsible for ensuring two important properties of transactions: *Atomicity* and *durability*. It ensures *atomicity* by undoing the actions of transactions that do not commit and *durability* by making sure that all actions of committed transactions survive **system crashes** (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

The recovery manager is one of the hardest components of a DBMS to design and implement. It must deal with a wide variety of database states because it is called on during system failures. In this chapter, we present the **ARIES** recovery algorithm, which is conceptually simple, works well with a wide range of concurrency control mechanisms, and is being used in an increasing number of database systems.

We begin with an introduction to ARIES in Section 18.1. We discuss the log, which a central data structure in recovery, in Section 18.2, and other recovery-related data structures in Section 18.3. We complete our coverage of recovery-related activity during normal processing by presenting the Write-Ahead Logging protocol in Section 18.4, and checkpointing in Section 18.5.

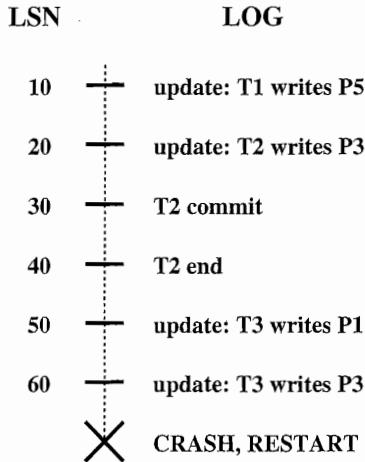
We discuss recovery from a crash in Section 18.6. Aborting (or rolling back) a single transaction is a special case of Undo, discussed in Section 18.6.3. We discuss media failures in Section 18.7, and conclude in Section 18.8 with a discussion of the interaction of concurrency control and recovery and other approaches to recovery. In this chapter, we consider recovery only in a centralized DBMS; recovery in a distributed DBMS is discussed in Chapter 22.

## 18.1 INTRODUCTION TO ARIES

**ARIES** is a recovery algorithm designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases:

1. **Analysis:** Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.
2. **Redo:** Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
3. **Undo:** Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

Consider the simple execution history illustrated in Figure 18.1. When the system is restarted, the Analysis phase identifies  $T_1$  and  $T_3$  as transactions



**Figure 18.1** Execution History with a Crash

active at the time of the crash and therefore to be undone;  $T_2$  as a committed transaction, and all its actions therefore to be written to disk; and  $P_1$ ,  $P_3$ , and  $P_5$  as potentially dirty pages. All the updates (including those of  $T_1$  and  $T_3$ ) are reapplied in the order shown during the Redo phase. Finally, the actions of  $T_1$  and  $T_3$  are undone in reverse order during the Undo phase; that is,  $T_3$ 's write of  $P_3$  is undone,  $T_3$ 's write of  $P_1$  is undone, and then  $T_1$ 's write of  $P_5$  is undone.

Three main principles lie behind the ARIES recovery algorithm:

- **Write-Ahead Logging:** Any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.
- **Repeating History During Redo:** On restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the time of the crash (effectively aborting them).
- **Logging Changes During Undo:** Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES from other recovery algorithms and is the basis for much of its simplicity and flexibility. In particular, ARIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The second and third points are also

**Crash Recovery:** IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all use a WAL scheme for recovery. IBM DB2 uses ARIES, and the others use schemes that are actually quite similar to ARIES (e.g., all changes are re-applied, not just the changes made by transactions that are ‘winners’) although there are several variations.

important in dealing with operations where redoing and undoing the operation are not exact inverses of each other. We discuss the interaction between concurrency control and crash recovery in Section 18.8, where we also discuss other approaches to recovery briefly.

## 18.2 THE LOG

The log, sometimes called the **trail** or **journal**, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more copies of the log on different disks (perhaps in different locations), so that the chance of all copies of the log being simultaneously lost is negligibly small.

The most recent portion of the log, called the **log tail**, is kept in main memory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every **log record** is given a unique *id* called the **log sequence number (LSN)**. As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in monotonically increasing order; this property is required for the ARIES recovery algorithm. If the log is a sequential file, in principle growing indefinitely, the LSN can simply be the address of the first byte of the log record.<sup>1</sup>

For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the **pageLSN**.

A log record is written for each of the following actions:

---

<sup>1</sup>In practice, various techniques are used to identify portions of the log that are ‘too old’ to be needed again to bound the amount of stable storage used for the log. Given such a bound, the log may be implemented as a ‘circular’ file, in which case the LSN may be the log record id plus a *wrap-count*.

- **Updating a Page:** After modifying the page, an *update* type record (described later in this section) is appended to the log tail. The pageLSN of the page is then set to the LSN of the update log record. (The page must be pinned in the buffer pool while these actions are carried out.)
- **Commit:** When a transaction decides to commit, it **force-writes** a *commit* type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record.<sup>2</sup> The transaction is considered to have committed at the instant that its commit log record is written to stable storage. (Some additional steps must be taken, e.g., removing the transaction's entry in the transaction table; these follow the writing of the commit log record.)
- **Abort:** When a transaction is aborted, an *abort* type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction (Section 18.6.3).
- **End:** As noted above, when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, an *end* type log record containing the transaction id is appended to the log.
- **Undoing an update:** When a transaction is rolled back (because the transaction is aborted, or during recovery from a crash), its updates are undone. When the action described by an update log record is undone, a *compensation log record*, or CLR, is written.

Every log record has certain fields: **prevLSN**, **transID**, and **type**. The set of all log records for a given transaction is maintained as a linked list going back in time, using the **prevLSN** field; this list must be updated whenever a log record is added. The **transID** field is the id of the transaction generating the log record, and the **type** field obviously indicates the type of the log record.

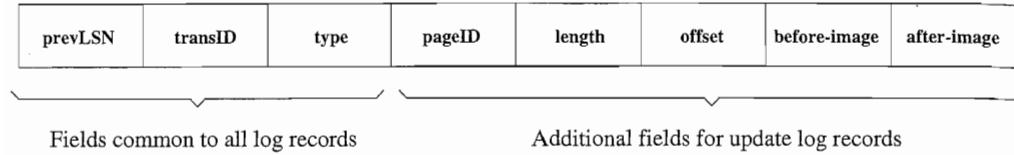
Additional fields depend on the type of the log record. We already mentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

## Update Log Records

The fields in an **update** log record are illustrated in Figure 18.2. The **pageID** field is the page id of the modified page; the length in bytes and the offset of the

---

<sup>2</sup>Note that this step requires the buffer manager to be able to selectively *force* pages to stable storage.



**Figure 18.2** Contents of an Update Log Record

change are also included. The **before-image** is the value of the changed bytes before the change; the **after-image** is the value after the change. An update log record that contains both before- and after-images can be used to redo the change and undo it. In certain contexts, which we do not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A **redo-only update** log record contains just the after-image; similarly an **undo-only update** record contains just the before-image.

## Compensation Log Records

A **compensation log record (CLR)** is written just before the change recorded in an update log record  $U$  is undone. (Such an undo can happen during normal system execution when a transaction is aborted or during recovery from a crash.) A compensation log record  $C$  describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. The compensation log record  $C$  also contains a field called **undoNextLSN**, which is the LSN of the next log record that is to be undone for the transaction that wrote update record  $U$ ; this field in  $C$  is set to the value of prevLSN in  $U$ .

As an example, consider the fourth update log record shown in Figure 18.3. If this update is undone, a CLR would be written, and the information in it would include the transID, pageID, length, offset, and before-image fields from the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-image value; thus, this value and the location of the affected bytes constitute the redo information for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 18.3.

Unlike an update log record, a CLR describes an action that will never be *undone*, that is, we never undo an undo action. The reason is simple: An update log record describes a change made by a transaction during normal execution and the transaction may subsequently be aborted, whereas a CLR describes an action taken to rollback a transaction for which the decision to abort has already been made. Therefore, the transaction *must* be rolled back, and the

undo action described by the CLR is definitely required. This observation is very useful because it bounds the amount of space needed for the log during restart from a crash: The number of CLRs that can be written during Undo is no more than the number of update log records for active transactions at the time of the crash.

A CLR may be written to stable storage (following WAL, of course) but the undo action it describes may not yet been written to disk when the system crashes again. In this case, the undo action described in the CLR is reapplied during the Redo phase, just like the action described in update log records.

For these reasons, a CLR contains the information needed to reapply, or redo, the change described but not to reverse it.

### 18.3 OTHER RECOVERY-RELATED STRUCTURES

In addition to the log, the following two tables contain important recovery-related information:

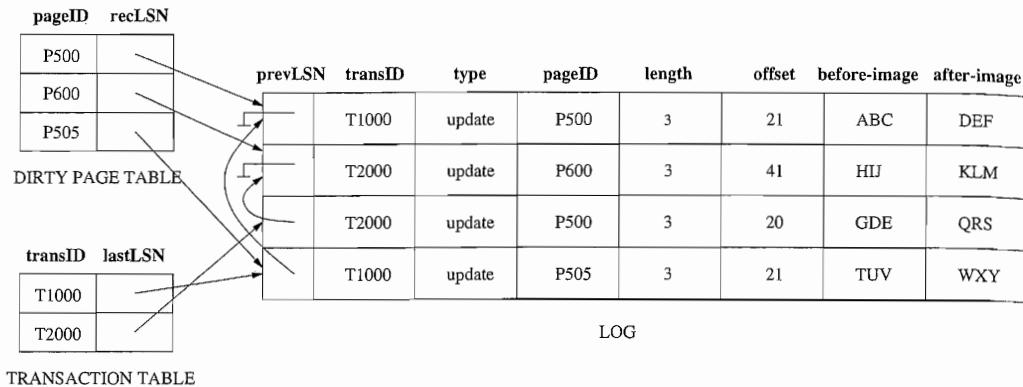
- **Transaction Table:** This table contains one entry for each active transaction. The entry contains (among other things) the transaction id, the status, and a field called **lastLSN**, which is the LSN of the most recent log record for this transaction. The **status** of a transaction can be that it is in progress, committed, or aborted. (In the latter two cases, the transaction will be removed from the table once certain ‘clean up’ steps are completed.)
- **Dirty page table:** This table contains one entry for each dirty page in the buffer pool, that is, each page with changes not yet reflected on disk. The entry contains a field **recLSN**, which is the LSN of the first log record that caused the page to become dirty. Note that this LSN identifies the earliest log record that might have to be redone for this page during restart from a crash.

During normal operation, these are maintained by the transaction manager and the buffer manager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the following simple example. Transaction  $T1000$  changes the value of bytes 21 to 23 on page  $P500$  from ‘ABC’ to ‘DEF’, transaction  $T2000$  changes ‘HIJ’ to ‘KLM’ on page  $P600$ , transaction  $T2000$  changes bytes 20 through 22 from ‘GDE’ to ‘QRS’ on page  $P500$ , then transaction  $T1000$  changes ‘TUV’ to ‘WXY’ on page  $P505$ . The dirty page table, the transaction table,<sup>3</sup> and

---

<sup>3</sup>The status field is not shown in the figure for space reasons; all transactions are in progress.



**Figure 18.3** Instance of Log and Transaction Table

the log at this instant are shown in Figure 18.3. Observe that the log is shown growing from top to bottom; older records are at the top. Although the records for each transaction are linked using the prevLSN field, the log as a whole also has a sequential order that is important—for example,  $T2000$ 's change to page  $P500$  follows  $T1000$ 's change to page  $P500$ , and in the event of a crash, these changes must be redone in the same order.

## 18.4 THE WRITE-AHEAD LOG PROTOCOL

Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage. This is accomplished by forcing all log records up to and including the one with LSN equal to the pageLSN to stable storage before writing the page to disk.

The importance of the WAL protocol cannot be overemphasized—WAL is the fundamental rule that ensures that a record of every change to the database is available while attempting to recover from a crash. If a transaction made a change and committed, the no-force approach means that some of these changes may not have been written to disk at the time of a subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crashes. Note that the definition of a *committed transaction* is effectively 'a transaction all of whose log records, including a commit record, have been written to stable storage'.

When a transaction is committed, the log tail is forced to stable storage, even if a no-force approach is being used. It is worth contrasting this operation with the actions taken under a force approach: If a force approach is used, all the pages modified by the transaction, rather than a portion of the log that includes all its records, must be forced to disk when the transaction commits. The set of

all changed pages is typically much larger than the log tail because the size of an update log record is close to (twice) the size of the changed bytes, which is likely to be much smaller than the page size. Further, the log is maintained as a sequential file, and all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is much smaller than the cost of writing all changed pages to disk.

## 18.5 CHECKPOINTING

A **checkpoint** is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash.

Checkpointing in ARIES has three steps. First, a **begin\_checkpoint** record is written to indicate when the checkpoint starts. Second, an **end\_checkpoint** record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log. The third step is carried out after the **end\_checkpoint** record is written to stable storage: A special **master** record containing the LSN of the *begin\_checkpoint* log record is written to a known place on stable storage. While the **end\_checkpoint** record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have is that the transaction table and dirty page table are accurate *as of the time of the begin\_checkpoint record*.

This kind of checkpoint, called a **fuzzy checkpoint**, is inexpensive because it does not require quiescing the system or writing out pages in the buffer pool (unlike some other forms of checkpointing). On the other hand, the effectiveness of this checkpointing technique is limited by the earliest recLSN of pages in the dirty pages table, because during restart we must redo changes starting from the log record whose LSN is equal to this recLSN. Having a background process that periodically writes dirty pages to disk helps to limit this problem.

When the system comes back up after a crash, the restart process begins by locating the most recent checkpoint record. For uniformity, the system always begins normal execution by taking a checkpoint, in which the transaction table and dirty page table are both empty.

## 18.6 RECOVERING FROM A SYSTEM CRASH

When the system is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 18.4.

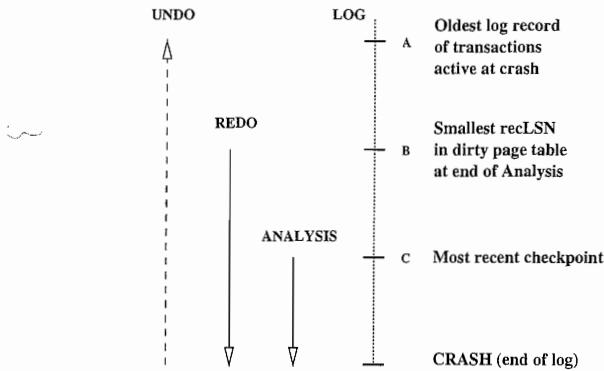


Figure 18.4 Three Phases of Restart in ARIES

The Analysis phase begins by examining the most recent begin\_checkpoint record, whose LSN is denoted *C* in Figure 18.4, and proceeds forward in the log until the last log record. The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash; this set of pages and the starting point for Redo (the smallest recLSN of any dirty page) are determined during Analysis. The Undo phase follows Redo and undoes the changes of all transactions active at the time of the crash; again, this set of transactions is identified during the Analysis phase. Note that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the most recent change first.

Observe that the relative order of the three points *A*, *B*, and *C* in the log may differ from that shown in Figure 18.4. The three phases of restart are described in more detail in the following sections.

### 18.6.1 Analysis Phase

The **Analysis** phase performs three tasks:

1. It determines the point in the log at which to start the Redo pass.
2. It determines (a conservative superset of the) pages in the buffer pool that were dirty at the time of the crash.
3. It identifies transactions that were active at the time of the crash and must be undone.

Analysis begins by examining the most recent begin\_checkpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end\_checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the time of the checkpoint.

(If additional log records are between the begin\_checkpoint and end\_checkpoint records, the tables must be adjusted to reflect the information in these records, but we omit the details of this step. See Exercise 18.9.) Analysis then scans the log in the forward direction until it reaches the end of the log:

- If an end log record for a transaction  $T$  is encountered,  $T$  is removed from the transaction table because it is no longer active.
- If a log record other than an end record for a transaction  $T$  is encountered, an entry for  $T$  is added to the transaction table if it is not already there. Further, the entry for  $T$  is modified:
  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a commit record, the status is set to C, otherwise it is set to U (indicating that it is to be undone).
- If a redoable log record affecting page  $P$  is encountered, and  $P$  is not in the dirty page table, an entry is inserted into this table with page id  $P$  and recLSN equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page  $P$  that may not have been written to disk.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the time of the crash—this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the time of the crash but may also contain some pages that were written to disk. If an *end-write* log record were written at the completion of each write operation, the dirty page table constructed during Analysis could be made more accurate, but in ARIES, the additional cost of writing *end-write* log records is not considered to be worth the gain.

As an example, consider the execution illustrated in Figure 18.3. Let us extend this execution by assuming that  $T2000$  commits, then  $T1000$  modifies another page, say,  $P700$ , and appends an update record to the log tail, and then the system crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in memory, are lost in the crash. The most recent checkpoint was taken at the beginning of the execution, with an empty transaction table and dirty page table; it is not shown in Figure 18.3. After examining this log record, which we assume is just before the first log record shown in the figure, Analysis initializes the two tables to be empty. Scanning forward in the log,  $T1000$  is added to the transaction table; in addition,  $P500$  is added to the dirty page table with recLSN equal to the LSN of the first shown log record. Similarly,  $T2000$  is added to the transaction table and  $P600$  is added to the dirty page table. There is no change based on the third log record, and the fourth record results in the addition of  $P505$  to

the dirty page table. The commit record for  $T2000$  (not in the figure) is now encountered, and  $T2000$  is removed from the transaction table.

The Analysis phase is now complete, and it is recognized that the only active transaction at the time of the crash is  $T1000$ , with lastLSN equal to the LSN of the fourth record in Figure 18.3. The dirty page table reconstructed in the Analysis phase is identical to that shown in the figure. The update log record for the change to  $P700$  is lost in the crash and not seen during the Analysis pass. Thanks to the WAL protocol, however, all is well—the corresponding change to page  $P700$  cannot have been written to disk either!

Some of the updates may have been written to disk; for concreteness, let us assume that the change to  $P600$  (and only this update) was written to disk before the crash. Therefore  $P600$  is not dirty, yet it is included in the dirty page table. The pageLSN on page  $P600$ , however, reflects the write because it is now equal to the LSN of the second update log record shown in Figure 18.3.

### 18.6.2 Redo Phase

During the **Redo** phase, ARIES reapplies the updates of *all* transactions, committed or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in the CLRs are also reapplied. This **repeating history** paradigm distinguishes ARIES from other proposed WAL-based recovery algorithms and causes the database to be brought to the same state it was in at the time of the crash.

The Redo phase begins with the log record that has the smallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that may not have been written to disk prior to the crash. Starting from this log record, Redo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action must be redone. The action must be redone unless one of the following conditions holds:

- The affected page is not in the dirty page table.
- The affected page is in the dirty page table, but the recLSN for the entry is *greater than* the LSN of the log record being checked.
- The pageLSN (stored on the page, which must be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously means that all changes to this page have been written to disk. Because the recLSN is the first update to this page that may

not have been written to disk, the second condition means that the update being checked was indeed propagated to disk. The third condition, which is checked last because it requires us to retrieve the page, also ensures that the update being checked was written to disk, because either this update or a later update to the page was written. (Recall our assumption that a write to a page is atomic; this assumption is important here!)

If the logged action must be redone:

1. The logged action is reapplied.
2. The pageLSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

Let us continue with the example discussed in Section 18.6.1. From the dirty page table, the smallest recLSN is seen to be the LSN of the first log record shown in Figure 18.3. Clearly, the changes recorded by earlier log records (there happen to be none in this example) have been written to disk. Now, Redo fetches the affected page,  $P500$ , and compares the LSN of this log record with the pageLSN on the page and, because we assumed that this page was not written to disk before the crash, finds that the pageLSN is less. The update is therefore reapplied; bytes 21 through 23 are changed to ‘DEF’, and the pageLSN is set to the LSN of this update log record.

Redo then examines the second log record. Again, the affected page,  $P600$ , is fetched and the pageLSN is compared to the LSN of the update log record. In this case, because we assumed that  $P600$  was written to disk before the crash, they are equal, and the update does not have to be redone.

The remaining log records are processed similarly, bringing the system back to the exact state it was in at the time of the crash. Note that the first two conditions indicating that a redo is unnecessary never hold in this example. Intuitively, they come into play when the dirty page table contains a very old recLSN, going back to before the most recent checkpoint. In this case, as Redo scans forward from the log record with this LSN, it encounters log records for pages that were written to disk prior to the checkpoint and therefore not in the dirty page table in the checkpoint. Some of these pages may be dirtied again after the checkpoint; nonetheless, the updates to these pages prior to the checkpoint need not be redone. Although the third condition alone is sufficient to recognize that these updates need not be redone, it requires us to fetch the affected page. The first two conditions allow us to recognize this situation without fetching the page. (The reader is encouraged to construct examples that illustrate the use of each of these conditions; see Exercise 18.8.)

At the end of the Redo phase, end type records are written for all transactions with status C, which are removed from the transaction table.

### 18.6.3 Undo Phase

The Undo phase, unlike the other two phases, scans backward from the end of the log. The goal of this phase is to undo the actions of all transactions active at the time of the crash, that is, to effectively abort them. This set of transactions is identified in the transaction table constructed by the Analysis phase.

#### The Undo Algorithm

Undo begins with the transaction table constructed by the Analysis phase, which identifies all transactions active at the time of the crash, and includes the LSN of the most recent log record (the lastLSN field) for each such transaction. Such transactions are called **loser transactions**. All actions of losers must be undone, and further, these actions must be undone in the reverse of the order in which they appear in the log.

Consider the set of lastLSN values for all loser transactions. Let us call this set **ToUndo**. Undo repeatedly chooses the largest (i.e., most recent) LSN value in this set and processes it, until ToUndo is empty. To process a log record:

1. If it is a CLR and the undoNextLSN value is not *null*, the undoNextLSN value is added to the set ToUndo; if the undoNextLSN is *null*, an end record is written for the transaction because it is completely undone, and the CLR is discarded.
2. If it is an update record, a CLR is written and the corresponding action is undone, as described in Section 18.2, and the prevLSN value in the update log record is added to the set ToUndo.

When the set ToUndo is empty, the Undo phase is complete. Restart is now complete, and the system can proceed with normal operations.

Let us continue with the scenario discussed in Sections 18.6.1 and 18.6.2. The only active transaction at the time of the crash was determined to be  $T1000$ . From the transaction table, we get the LSN of its most recent log record, which is the fourth update log record in Figure 18.3. The update is undone, and a CLR is written with undoNextLSN equal to the LSN of the first log record in the figure. The next record to be undone for transaction  $T1000$  is the first log record in the figure. After this is undone, a CLR and an end log record for  $T1000$  are written, and the Undo phase is complete.

In this example, undoing the action recorded in the first log record causes the action of the third log record, which is due to a committed transaction, to be overwritten and thereby lost! This situation arises because  $T'2000$  overwrote a data item written by  $T1000$  while  $T1000$  was still active; if Strict 2PL were followed,  $T'2000$  would not have been allowed to overwrite this data item.

## Aborting a Transaction

Aborting a transaction is just a special case of the Undo phase of Restart in which a single transaction, rather than a set of transactions, is undone. The example in Figure 18.5, discussed next, illustrates this point.

## Crashes during Restart

It is important to understand how the Undo algorithm presented in Section 18.6.3 handles repeated system crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we discuss Undo in the presence of system crashes using an execution history, shown in Figure 18.5, that abstracts away unnecessary detail. This example illustrates how aborting a transaction is a special case of Undo and how the use of CLRs ensures that the Undo action for an update log record is not applied twice.

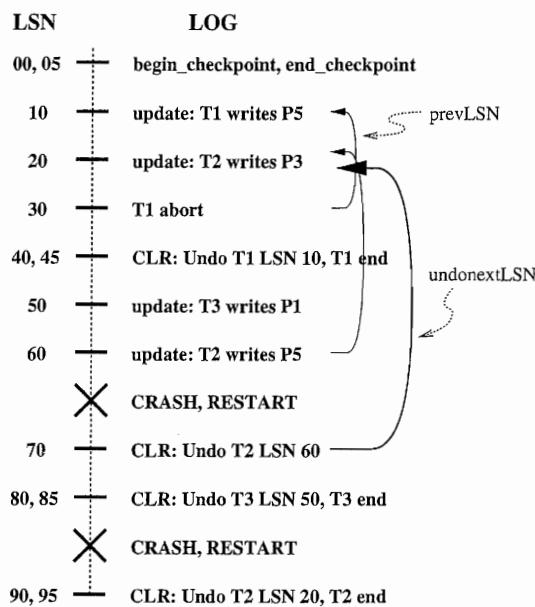


Figure 18.5 Example of Undo with Repeated Crashes

The log shows the order in which the DBMS executed various actions; note that the LSNs are in ascending order, and that each log record for a transaction has a prevLSN field that points to the previous log record for that transaction. We have not shown *null* prevLSNs, that is, some special value used in the prevLSN field of the first log record for a transaction to indicate that there is no previous log record. We also compacted the figure by occasionally displaying two log records (separated by a comma) on a single line.

Log record (with LSN) 30 indicates that  $T_1$  aborts. All actions of this transaction should be undone in reverse order, and the only action of  $T_1$ , described by the update log record 10, is indeed undone as indicated by CLR 40.

After the first crash, Analysis identifies  $P_1$  (with recLSN 50),  $P_3$  (with recLSN 20), and  $P_5$  (with recLSN 10) as dirty pages. Log record 45 shows that  $T_1$  is a completed transaction; hence, the transaction table identifies  $T_2$  (with lastLSN 60) and  $T_3$  (with lastLSN 50) as active at the time of the crash. The Redo phase begins with log record 10, which is the minimum recLSN in the dirty page table, and reapplys all actions (for the update and CLR records), as per the Redo algorithm presented in Section 18.6.2.

The ToUndo set consists of LSNs 60, for  $T_2$ , and 50, for  $T_3$ . The Undo phase now begins by processing the log record with LSN 60 because 60 is the largest LSN in the ToUndo set. The update is undone, and a CLR (with LSN 70) is written to the log. This CLR has undoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for  $T_2$ . Now the largest remaining LSN in the ToUndo set is 50. The write corresponding to log record 50 is now undone, and a CLR describing the change is written. This CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction  $T_3$ . Therefore  $T_3$  is completely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the system crashes a second time; however, the changes described by these records may not have been written to disk.

When the system is restarted after the second crash, Analysis determines that the only active transaction at the time of the crash was  $T_2$ ; in addition, the dirty page table is identical to what it was during the previous restart. Log records 10 through 85 are processed again during Redo. (If some of the changes made during the previous Redo were written to disk, the pageLSNs on the affected pages are used to detect this situation and avoid writing these pages again.) The Undo phase considers the only LSN in the ToUndo set, 70, and processes it by adding the undoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed by undoing  $T_2$ 's write of page  $P_3$ , and a CLR is written (LSN 90). Because 20 is the first of  $T_2$ 's log records—and therefore, the last of its records

to be undone—the undoNextLSN field in this CLR is *null*, an end record is written for  $T_2$ , and the ToUndo set is now empty.

Recovery is now complete, and normal execution can resume with the writing of a checkpoint record.

This example illustrated repeated crashes during the Undo phase. For completeness, let us consider what happens if the system crashes while Restart is in the Analysis or Redo phase. If a crash occurs during the Analysis phase, all the work done in this phase is lost, and on restart the Analysis phase starts afresh with the same information as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that some of the changes made during Redo may have been written to disk prior to the crash. Restart starts again with the Analysis phase and then the Redo phase, and some update log records that were redone the first time around will not be redone a second time because the pageLSN is now equal to the update record's LSN (although the pages have to be fetched again to detect this).

We can take checkpoints during Restart to minimize repeated work in the event of a crash, but we do not discuss this point.

## 18.7 MEDIA RECOVERY

Media recovery is based on periodically making a copy of the database. Because copying a large database object such as a file can take a long time, and the DBMS must be allowed to continue with its operations in the meantime, creating a copy is handled in a manner similar to taking a fuzzy checkpoint.

When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of committed transactions and undo the changes of uncommitted transactions (as of the time of the media recovery operation).

The begin\_checkpoint LSN of the most recent complete checkpoint is recorded along with the copy of the database object to minimize the work in reapplying changes of committed transactions. Let us compare the smallest recLSN of a dirty page in the corresponding end\_checkpoint record with the LSN of the begin\_checkpoint record and call the smaller of these two LSNs  $I$ . We observe that the actions recorded in all log records with LSNs less than  $I$  must be reflected in the copy. Thus, only log records with LSNs greater than  $I$  need be reapplied to the copy.

Finally, the updates of transactions that are incomplete at the time of media recovery or that were aborted after the fuzzy copy was completed need to be undone to ensure that the page reflects only the actions of committed transactions. The set of such transactions can be identified as in the Analysis pass, and we omit the details.

## 18.8 OTHER APPROACHES AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the most popular alternative recovery algorithms also maintain a log of database actions according to the WAL protocol. A major distinction between ARIES and these variants is that the Redo phase in ARIES *repeats history*, that is, redoes the actions of *all* transactions, not just the non-losers. Other algorithms redo only the non-losers, and the Redo phase follows the Undo phase, in which the actions of losers are rolled back.

Thanks to the repeating history paradigm and the use of CLRs, ARIES supports fine-granularity locks (record-level locks) and logging of logical operations rather than just byte-level modifications. For example, consider a transaction  $T$  that inserts a data entry 15\* into a B+ tree index. Between the time this insert is done and the time that  $T$  is eventually aborted, other transactions may also insert and delete entries from the tree. If record-level locks are set rather than page-level locks, the entry 15\* may be on a different physical page when  $T$  aborts from the one that  $T$  inserted it into. In this case, the undo operation for the insert of 15\* must be recorded in logical terms because the physical (byte-level) actions involved in undoing this operation are not the inverse of the physical actions involved in inserting the entry.

Logging logical operations yields considerably higher concurrency, although the use of fine-granularity locks can lead to increased locking activity (because more locks must be set). Hence, there is a trade-off between different WAL-based recovery schemes. We chose to cover ARIES because it has several attractive properties, in particular, its simplicity and its ability to support fine-granularity locks and logging of logical operations.

One of the earliest recovery algorithms, used in the System R prototype at IBM, takes a very different approach. There is no logging and, of course, no WAL protocol. Instead, the database is treated as a collection of pages and accessed through a **page table**, which maps page ids to disk addresses. When a transaction makes changes to a data page, it actually makes a copy of the page, called the **shadow** of the page, and changes the shadow page. The transaction copies the appropriate part of the page table and changes the entry for the changed page to point to the shadow, so that it can see the

changes; however, other transactions continue to see the original page table, and therefore the original page, until this transaction commits. Aborting a transaction is simple: Just discard its shadow versions of the page table and the data pages. Committing a transaction involves making its version of the page table public and discarding the original data pages that are superseded by shadow pages.

This scheme suffers from a number of problems. First, data becomes highly fragmented due to the replacement of pages by shadow versions, which may be located far from the original page. This phenomenon reduces data clustering and makes good garbage collection imperative. Second, the scheme does not yield a sufficiently high degree of concurrency. Third, there is a substantial storage overhead due to the use of shadow pages. Fourth, the process aborting a transaction can itself run into deadlocks, and this situation must be specially handled because the semantics of aborting an abort transaction gets murky.

For these reasons, even in System R, shadow paging was eventually superseded by WAL-based recovery techniques.

## 18.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the advantages of the ARIES recovery algorithm? (**Section 18.1**)
- Describe the three steps in crash recovery in ARIES? What is the goal of the Analysis phase? The redo phase? The undo phase? (**Section 18.1**)
- What is the LSN of a log record? (**Section 18.2**)
- What are the different types of log records and when are they written? (**Section 18.2**)
- What information is maintained in the transaction table and the dirty page table? (**Section 18.3**)
- What is Write-Ahead Logging? What is forced to disk at the time a transaction commits? (**Section 18.4**)
- What is a fuzzy checkpoint? Why is it useful? What is a master log record? (**Section 18.5**)
- In which direction does the Analysis phase of recovery scan the log? At which point in the log does it begin and end the scan? (**Section 18.6.1**)
- Describe what information is gathered in the Analysis phase and how. (**Section 18.6.1**)

- In which direction does the Redo phase of recovery process the log? At which point in the log does it begin and end? (**Section 18.6.2**)
- What is a redoable log record? Under what conditions is the logged action redone? What steps are carried out when a logged action is redone? (**Section 18.6.2**)
- In which direction does the Undo phase of recovery process the log? At which point in the log does it begin and end? (**Section 18.6.3**)
- What are loser transactions? How are they processed in the Undo phase and in what order? (**Section 18.6.3**)
- Explain what happens if there are crashes during the Undo phase of recovery. What is the role of CLRs? What if there are crashes during the Analysis and Redo phases? (**Section 18.6.3**)
- How does a DBMS recover from media failure without reading the complete log? (**Section 18.7**)
- Record-level logging increases concurrency. What are the potential problems, and how does ARIES address them? (**Section 18.8**)
- What is shadow paging? (**Section 18.8**)

## EXERCISES

**Exercise 18.1** Briefly answer the following questions:

1. How does the recovery manager ensure atomicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a system crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

**Exercise 18.2** Briefly answer the following questions:

1. What are the properties required of LSNs?
2. What are the fields in an update log record? Explain the use of each field.
3. What are redoable log records?
4. What are the differences between update log records and CLRs?

**Exercise 18.3** Briefly answer the following questions:

1. What are the roles of the Analysis, Redo, and Undo phases in ARIES?
2. Consider the execution shown in Figure 18.6.

LSN	LOG
00	begin_checkpoint
10	end_checkpoint
20	update: T1 writes P5
30	update: T2 writes P3
40	T2 commit
50	T2 end
60	update: T3 writes P3
70	T1 abort
X	CRASH, RESTART

Figure 18.6 Execution with a Crash

LSN	LOG
00	update: T1 writes P2
10	update: T1 writes P1
20	update: T2 writes P5
30	update: T3 writes P3
40	T3 commit
50	update: T2 writes P5
60	update: T2 writes P3
70	T2 abort

Figure 18.7 Aborting a Transaction

- (a) What is done during Analysis? (Be precise about the points at which Analysis begins and ends and describe the contents of any tables constructed in this phase.)
- (b) What is done during Redo? (Be precise about the points at which Redo begins and ends.)
- (c) What is done during Undo? (Be precise about the points at which Undo begins and ends.)

**Exercise 18.4** Consider the execution shown in Figure 18.7.

1. Extend the figure to show prevLSN and undonextLSN values.
2. Describe the actions taken to rollback transaction  $T2$ .

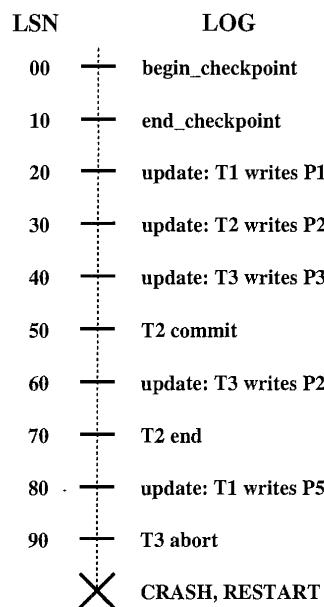


Figure 18.8 Execution with Multiple Crashes

3. Show the log after  $T_2$  is rolled back, including all prevLSN and undonextLSN values in log records.

**Exercise 18.5** Consider the execution shown in Figure 18.8. In addition, the system crashes during recovery after writing two log records to stable storage and again after writing another two log records.

1. What is the value of the LSN stored in the master log record?
2. What is done during Analysis?
3. What is done during Redo?
4. What is done during Undo?
5. Show the log when recovery is complete, including all non-null prevLSN and undonextLSN values in log records.

**Exercise 18.6** Briefly answer the following questions:

1. How is checkpointing done in ARIES?
2. Checkpointing can also be done as follows: Quiesce the system so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and transaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointing approach of ARIES?
3. What happens if a second begin\_checkpoint record is encountered during the Analysis phase?
4. Can a second end\_checkpoint record be encountered during the Analysis phase?
5. Why is the use of CLRs important for the use of undo actions that are not the physical inverse of the original update?

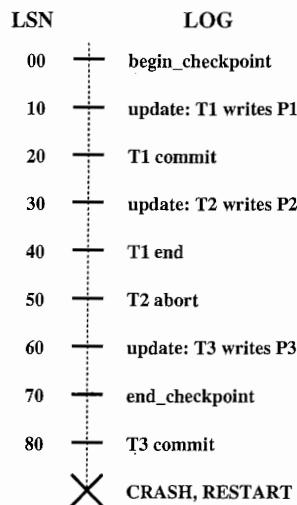


Figure 18.9 Log Records between Checkpoint Records

6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

**Exercise 18.7** Briefly answer the following questions:

1. If the system fails repeatedly during recovery, what is the maximum number of log records that can be written (as a function of the number of update and other log records written before the crash) before restart completes successfully?
2. What is the oldest log record we need to retain?
3. If a bounded amount of stable storage is used for the log, how can we always ensure enough stable storage to hold all log records written during restart?

**Exercise 18.8** Consider the three conditions under which a redo is unnecessary (Section 20.2.2).

1. Why is it cheaper to test the first two conditions?
2. Describe an execution that illustrates the use of the first condition.
3. Describe an execution that illustrates the use of the second condition.

**Exercise 18.9** The description in Section 18.6.1 of the Analysis phase made the simplifying assumption that no log records appeared between the begin\_checkpoint and end\_checkpoint records for the most recent complete checkpoint. The following questions explore how such records should be handled.

1. Explain why log records could be written between the begin\_checkpoint and end\_checkpoint records.
2. Describe how the Analysis phase could be modified to handle such records.
3. Consider the execution shown in Figure 18.9. Show the contents of the end\_checkpoint record.
4. Illustrate your modified Analysis phase on the execution shown in Figure 18.9.



# Chapter 6

## Experimental Design

This chapter contains the book chapters:

D. Lilja. Measuring Computer Performance: A Practitioner's Guide. Chapters 1, 2, and 6, pp. 1–24 and 82–107 (50 of 261). Cambridge University Press, 2000. ISBN: 978-0-521-64105-0

Properly comparing different designs and implementations of a given service or system is a hard problem. A wealth of techniques have been developed to measure or estimate the performance of computer systems, including analytical modeling, simulation, and experimentation. Use of these techniques requires care in assumption documentation, metric selection, benchmark design, measurement approach, and experimental setup. While a comprehensive treatment is beyond the scope of ACS, we intend to provide an overview of the issues, as well as measurement exercises to develop competence in structuring experiments. *The ultimate goal of this portion of the material is to provide us with basic concepts in measurement of computer systems, as well as allow us to structure and carry out experiments to measure basic performance properties of a system under study.*

The learning goals for this portion of the material are listed below.

- Explain the three main methodologies for performance measurement and modeling: analytical modeling, simulation, and experimentation.
- Design and execute experiments to measure the performance of a system.

*Chapter 6 of Lilja's book is given to deepen understanding of available measurement strategies; however, it is to be considered as an additional reading and not fundamental to the attainment of the learning goals above.*

# 1

# Introduction

'Performance can be bad, but can it ever be wrong?'

*Jim Kohn, SGI/Cray Research, Inc.*

## 1.1 Measuring performance

If the automobile industry had followed the same development cycles as the computer industry, it has been speculated that a Rolls Royce car would cost less than \$100 with an efficiency of more than 200 miles per gallon of gasoline. While we certainly get more car for our money now than we did twenty years ago, no other industry has ever changed at the incredible rate of the computer and electronics industry.

Computer systems have gone from being the exclusive domain of a few scientists and engineers who used them to speed up some esoteric computations, such as calculating the trajectory of artillery shells, for instance, to being so common that they go unnoticed. They have replaced many of the mechanical control systems in our cars, thereby reducing cost while improving efficiency, reliability, and performance. They have made possible such previously science-fiction-like devices as cellular phones. They have provided countless hours of entertainment for children ranging in age from one to one hundred. They have even brought sound to the common greeting card. One constant throughout this proliferation and change, however, has been the need for system developers and users to understand the *performance* of these computer-based devices.

While measuring the cost of a system is usually relatively straightforward (except for the confounding effects of manufacturers' discounts to special customers), determining the performance of a computer system can oftentimes seem like an exercise in futility. Surprisingly, one of the main difficulties in measuring performance is that reasonable people often disagree strongly on how performance should be measured or interpreted, and even on what 'performance' actually means.

*Performance analysis* as applied to experimental computer science and engineering should be thought of as a combination of *measurement*, *interpretation*, and *communication* of a computer system's 'speed' or 'size' (sometimes referred to as its 'capacity'). The terms speed and size are quoted in this context to emphasize that their actual definitions often depend on the specifics of the situation. Also, it is important to recognize that we need not necessarily be dealing with complete systems. Often it is necessary to analyze only a small portion of the system independent of the other components. For instance, we may be interested in studying the performance of a certain computer system's network interface independent of the size of its memory or the type of processor. Unfortunately, the components of a computer system can interact in incredibly complex, and frequently unpredictable, ways. One of the signs of an expert computer performance analyst is that he or she can tease apart these interactions to determine the performance effect due only to a particular component.

One of the most interesting tasks of the performance analyst can be figuring out how to measure the necessary data. A large dose of creativity may be needed to develop good measurement techniques that perturb the system as little as possible while providing accurate, reproducible results. After the necessary data have been gathered, the analyst must interpret the results using appropriate statistical techniques. Finally, even excellent measurements interpreted in a statistically appropriate fashion are of no practical use to anyone unless they are communicated in a clear and consistent manner.

---

## 1.2 Common goals of performance analysis

---

The goals of any analysis of the performance of a computer system, or one of its components, will depend on the specific situation and the skills, interests, and abilities of the analyst. However, we can identify several different typical goals of performance analysis that are useful both to computer-system designers and to users.

- **Compare alternatives.** When purchasing a new computer system, you may be confronted with several different systems from which to choose. Furthermore, you may have several different options within each system that may impact both cost and performance, such as the size of the main memory, the number of processors, the type of network interface, the size and number of disk drives, the type of system software (i.e., the operating system and compilers), and on and on. The goal of the performance analysis task in this case is to provide quantitative information about which configurations are best under specific conditions.

- **Determine the impact of a feature.** In designing new systems, or in upgrading existing systems, you often need to determine the impact of adding or removing a specific feature of the system. For instance, the designer of a new processor may want to understand whether it makes sense to add an additional floating-point execution unit to the microarchitecture, or whether the size of the on-chip cache should be increased instead. This type of analysis is often referred to as a *before-and-after* comparison since only one well-defined component of the system is changed.
- **System tuning.** The goal of performance analysis in system tuning is to find the set of parameter values that produces the best overall performance. In time-shared operating systems, for instance, it is possible to control the number of processes that are allowed to actively share the processor. The overall performance perceived by the system users can be substantially impacted both by this number, and by the time quantum allocated to each process. Many other system parameters, such as disk and network buffer sizes, for example, can also significantly impact the system performance. Since the performance impacts of these various parameters can be closely interconnected, finding the best set of parameter values can be a very difficult task.
- **Identify relative performance.** The performance of a computer system typically has meaning only in the context of its performance relative to something else, such as another system or another configuration of the same system. The goal in this situation may be to quantify the change in performance relative to history – that is, relative to previous generations of the system. Another goal may be to quantify the performance relative to a customer’s expectations, or to a competitor’s systems, for instance.
- **Performance debugging.** Debugging a program for correct execution is a fundamental prerequisite for any application program. Once the program is functionally correct, however, the performance analysis task becomes one of finding performance problems. That is, the program now produces the correct results, but it may be much slower than desired. The goal of the performance analyst at this point is to apply the appropriate tools and analysis techniques to determine why the program is not meeting performance expectations. Once the performance problems are identified, they can, it is to be hoped, be corrected.
- **Set expectations.** Users of computer systems may have some idea of what the capabilities of the next generation of a line of computer systems should be. The task of the performance analyst in this case is to set the appropriate expectations for what a system is actually capable of doing.

In all of these situations, the effort involved in the performance-analysis task should be proportional to the cost of making the wrong decision. For example, if

you are comparing different manufacturers' systems to determine which best satisfies the requirements for a large purchasing decision, the financial cost of making the wrong decision could be quite substantial, both in terms of the cost of the system itself, and in terms of the subsequent impacts on the various parts of a large project or organization. In this case, you will probably want to perform a very detailed, thorough analysis. If, however, you are simply trying to choose a system for your own personal use, the cost of choosing the wrong one is minimal. Your performance analysis in this case may be correctly limited to reading a few reviews from a trade magazine.

### **1.3 Solution techniques**

When one is confronted with a performance-analysis problem, there are three fundamental techniques that can be used to find the desired solution. These are *measurements* of existing systems, *simulation*, and *analytical modeling*. Actual measurements generally provide the best results since, given the necessary measurement tools, no simplifying assumptions need to be made. This characteristic also makes results based on measurements of an actual system the most believable when they are presented to others. Measurements of real systems are not very flexible, however, in that they provide information about only the specific system being measured. A common goal of performance analysis is to characterize how the performance of a system changes as certain parameters are varied. In an actual system, though, it may be very difficult, if not impossible, to change some of these parameters. Evaluating the performance impact of varying the speed of the main memory system, for instance, is simply not possible in most real systems. Furthermore, measuring some aspects of performance on an actual system can be very time-consuming and difficult. Thus, while measurements of real systems may provide the most compelling results, their inherent difficulties and limitations produce a need for other solution techniques.

A simulation of a computer system is a program written to model important features of the system being analyzed. Since the simulator is nothing more than a program, it can be easily modified to study the impact of changes made to almost any of the simulated components. The cost of a simulation includes both the time and effort required to write and debug the simulation program, and the time required to execute the necessary simulations. Depending on the complexity of the system being simulated, and the level of detail at which it is modeled, these costs can be relatively low to moderate compared with the cost of purchasing a real machine on which to perform the corresponding experiments.

The primary limitation of a simulation-based performance analysis is that it is impossible to model every small detail of the system being studied. Consequently,

simplifying assumptions are required in order to make it possible to write the simulation program itself, and to allow it to execute in a reasonable amount of time. These simplifying assumptions then limit the accuracy of the results by lowering the fidelity of the model compared with how an actual system would perform. Nevertheless, simulation enjoys tremendous popularity for computer-systems analysis due to its high degree of flexibility and its relative ease of implementation.

The third technique in the performance analyst's toolbox is analytical modeling. An analytical model is a mathematical description of the system. Compared with a simulation or a measurement of a real machine, the results of an analytical model tend to be much less believable and much less accurate. A simple analytical model, however, can provide some quick insights into the overall behavior of the system, or one of its components. This insight can then be used to help focus a more detailed measurement or simulation experiment. Analytical models are also useful in that they provide at least a coarse level of validation of a simulation or measurement. That is, an analytical model can help confirm whether the results produced by a simulator, or the values measured on a real system, appear to be reasonable.

**Example.** The delay observed by an application program when accessing memory can have a significant impact on its overall execution time. Direct measurements of this time on a real machine can be quite difficult, however, since the detailed steps involved in the operation of a complex memory hierarchy structure are typically not observable from a user's application program. A sophisticated user may be able to write simple application programs that exercise specific portions of the memory hierarchy to thereby *infer* important memory-system parameters. For instance, the execution time of a simple program that repeatedly references the same variable can be used to estimate the time required to access the first-level cache. Similarly, a program that always forces a cache miss can be used to indirectly measure the main memory access time. Unfortunately, the impact of these system parameters on the execution time of a complete application program is very dependent on the precise memory-referencing characteristics of the program, which can be difficult to determine.

Simulation, on the other hand, is a powerful technique for studying memory-system behavior due to its high degree of flexibility. Any parameter of the memory, including the cache associativity, the relative cache and memory delays, the sizes of the cache and memory, and so forth, can be easily changed to study its impact on performance. It can be challenging, however, to accurately model in a simulator the overlap of memory delays and the execution of other instructions in contemporary processors that incorporate such performance-enhancing features as out-of-order instruction issuing, branch prediction, and nonblocking caches. Even with the necessary simplifying assumptions, the results of a detailed

simulation can still provide useful insights into the effect of the memory system on the performance of a specific application program.

Finally, a simple analytical model of the memory system can be developed as follows. Let  $t_c$  be the time delay observed by a memory reference if the memory location being referenced is in the cache. Also, let  $t_m$  be the corresponding delay if the referenced location is not in the cache. The cache *hit ratio*, denoted  $h$ , is the fraction of all memory references issued by the processor that are satisfied by the cache. The fraction of references that *miss* in the cache and so must also access the memory is  $1 - h$ . Thus, the average time required for all cache hits is  $ht_c$  while the average time required for all cache misses is  $(1 - h)t_m$ . A simple model of the overall average memory-access time observed by an executing program then is

$$t_{\text{avg}} = ht_c + (1 - h)t_m. \quad (1.1)$$

To apply this simple model to a specific application program, we would need to know the hit ratio,  $h$ , for the program, and the values of  $t_c$  and  $t_m$  for the system. These memory-access-time parameters,  $t_c$  and  $t_m$ , may often be found in the manufacturer's specifications of the system. Or, they may be inferred through a measurement, as described above and as explored further in the exercises in Chapter 6. The hit ratio,  $h$ , for an application program is typically more difficult to obtain. It is often found through a simulation of the application, though. Although this model will provide only a very coarse estimate of the average memory-access time observed by a program, it can provide us with some insights into the relative effects of increasing the hit ratio, or changing the memory-timing parameters, for instance. ◇

The key differences among these solution techniques are summarized in Table 1.1. The *flexibility* of a technique is an indication of how easy it is to change the system to study different configurations. The *cost* corresponds to the time, effort, and money necessary to perform the appropriate experiments using each technique. The *believability* of a technique is high if a knowledgeable individual has a high degree of confidence that the result produced using that technique is likely to be correct in practice. It is much easier for someone to believe that the execution time of a given application program will be within a certain range when you can demonstrate it on an actual machine, for instance, than when relying on a mere simulation. Similarly, most people are more likely to believe the results of a simulation study than one that relies entirely on an analytical model. Finally, the *accuracy* of a solution technique indicates how closely results obtained when using that technique correspond to the results that would have been obtained on a real system.

The choice of a specific solution technique depends on the problem being solved. One of the skills that must be developed by a computer-systems performance analyst is determining which technique is the most appropriate for the

**Table 1.1.** A comparison of the performance-analysis solution techniques

Characteristic	Solution technique		
	Analytical modeling	Simulation	Measurement
Flexibility	High	High	Low
Cost	Low	Medium	High
Believability	Low	Medium	High
Accuracy	Low	Medium	High

given situation. The following chapters are designed to help you develop precisely this skill.

## 1.4 Summary

Computer-systems performance analysis often feels more like an art than a science. Indeed, different individuals can sometimes reach apparently contradictory conclusions when analyzing the same system or set of systems. While this type of ambiguity can be quite frustrating, it is often due to misunderstandings of what was actually being measured, or disagreements about how the data should be analyzed or interpreted. These differences further emphasize the need to clearly communicate all results and to completely specify the tools, techniques, and system parameters used to collect and understand the data. As you study the following chapters, my hope is that you will begin to develop an appreciation for this art of measurement, interpretation, and communication in addition to developing a deeper understanding of its mathematical and scientific underpinnings.

## 1.5 Exercises

1. Respond to the question quoted at the beginning of this chapter, ‘Performance can be bad, but can it ever be wrong?’
2. Performance analysis should be thought of as a decision-making process. Section 1.2 lists several common goals of a performance-analysis experiment. List other possible goals of the performance-analysis decision-making process. Who are the beneficiaries of each of these possible goals?

3. Table 1.1 compares the three main performance-analysis solution techniques across several criteria. What additional criteria could be used to compare these techniques?
4. Identify the most appropriate solution technique for each of the following situations.
  - (a) Estimating the performance benefit of a new feature that an engineer is considering adding to a computer system currently being designed.
  - (b) Determining when it is time for a large insurance company to upgrade to a new system.
  - (c) Deciding the best vendor from which to purchase new computers for an expansion to an academic computer lab.
  - (d) Determining the minimum performance necessary for a computer system to be used on a deep-space probe with very limited available electrical power.

## 2 Metrics of performance

‘Time is a great teacher, but unfortunately it kills all its pupils.’

Hector Berlioz

### 2.1 What is a performance metric?

Before we can begin to understand any aspect of a computer system’s performance, we must determine what things are interesting and useful to measure. The basic characteristics of a computer system that we typically need to measure are:

- a *count* of how many times an event occurs,
- the *duration* of some time interval, and
- the *size* of some parameter.

For instance, we may need to count how many times a processor initiates an input/output request. We may also be interested in how long each of these requests takes. Finally, it is probably also useful to determine the number of bits transmitted and stored.

From these types of measured values, we can derive the actual value that we wish to use to describe the performance of the system. This value is called a *performance metric*.

If we are interested specifically in the time, count, or size value measured, we can use that value directly as our performance metric. Often, however, we are interested in normalizing event counts to a common time basis to provide a speed metric such as operations executed per second. This type of metric is called a *rate metric* or *throughput* and is calculated by dividing the count of the number of events that occur in a given interval by the time interval over which the events occur. Since a rate metric is normalized to a common time basis, such as seconds, it is useful for comparing different measurements made over different time intervals.

Choosing an appropriate performance metric depends on the goals for the specific situation and the cost of gathering the necessary information. For

example, suppose that you need to choose between two different computer systems to use for a short period of time for one specific task, such as choosing between two systems to do some word processing for an afternoon. Since the penalty for being wrong in this case, that is, choosing the slower of the two machines, is very small, you may decide to use the processors' clock frequencies as the performance metric. Then you simply choose the system with the fastest clock. However, since the clock frequency is not a reliable performance metric (see Section 2.3.1), you would want to choose a better metric if you are trying to decide which system to buy when you expect to purchase hundreds of systems for your company. Since the consequences of being wrong are much larger in this case (you could lose your job, for instance!), you should take the time to perform a rigorous comparison using a better performance metric. This situation then begs the question of what constitutes a good performance metric.

---

## 2.2 Characteristics of a good performance metric

---

There are many different metrics that have been used to describe a computer system's performance. Some of these metrics are commonly used throughout the field, such as MIPS and MFLOPS (which are defined later in this chapter), whereas others are invented for new situations as they are needed. Experience has shown that not all of these metrics are 'good' in the sense that sometimes using a particular metric can lead to erroneous or misleading conclusions. Consequently, it is useful to understand the characteristics of a 'good' performance metric. This understanding will help when deciding which of the existing performance metrics to use for a particular situation, and when developing a new performance metric.

A performance metric that satisfies all of the following requirements is generally useful to a performance analyst in allowing accurate and detailed comparisons of different measurements. These criteria have been developed by observing the results of numerous performance analyses over many years. While they should not be considered absolute requirements of a performance metric, it has been observed that using a metric that does not satisfy these requirements can often lead the analyst to make erroneous conclusions.

- 1. Linearity.** Since humans intuitively tend to think in linear terms, the value of the metric should be linearly proportional to the actual performance of the machine. That is, if the value of the metric changes by a certain ratio, the actual performance of the machine should change by the same ratio. This proportionality characteristic makes the metric intuitively appealing to most people. For example, suppose that you are upgrading your system to a system

whose speed metric (i.e. execution-rate metric) is twice as large as the same metric on your current system. You then would expect the new system to be able to run your application programs in half the time taken by your old system. Similarly, if the metric for the new system were three times larger than that of your current system, you would expect to see the execution times reduced to one-third of the original values.

Not all types of metrics satisfy this proportionally requirement. Logarithmic metrics, such as the dB scale used to describe the intensity of sound, for example, are nonlinear metrics in which an increase of one in the value of the metric corresponds to a factor of ten increase in the magnitude of the observed phenomenon. There is nothing inherently wrong with these types of nonlinear metrics, it is just that linear metrics tend to be more intuitively appealing when interpreting the performance of computer systems.

2. **Reliability.** A performance metric is considered to be *reliable* if system A always outperforms system B when the corresponding values of the metric for both systems indicate that system A should outperform system B. For example, suppose that we have developed a new performance metric called WIPS that we have designed to compare the performance of computer systems when running the class of word-processing application programs. We measure system A and find that it has a WIPS rating of 128, while system B has a WIPS rating of 97. We then can say that WIPS is a reliable performance metric for word-processing application programs if system A always outperforms system B when executing these types of applications.

While this requirement would seem to be so obvious as to be unnecessary to state explicitly, several commonly used performance metrics do not in fact satisfy this requirement. The MIPS metric, for instance, which is described further in Section 2.3.2, is notoriously unreliable. Specifically, it is not unusual for one processor to have a higher MIPS rating than another processor while the second processor actually executes a specific program in less time than does the processor with the higher value of the metric. Such a metric is essentially useless for summarizing performance, and we say that it is unreliable.

3. **Repeatability.** A performance metric is *repeatable* if the same value of the metric is measured each time the same experiment is performed. Note that this also implies that a good metric is deterministic.
4. **Easiness of measurement.** If a metric is not easy to measure, it is unlikely that anyone will actually use it. Furthermore, the more difficult a metric is to measure directly, or to derive from other measured values, the more likely

it is that the metric will be determined incorrectly. The only thing worse than a bad metric is a metric whose value is measured incorrectly.

5. **Consistency.** A *consistent* performance metric is one for which the units of the metric and its precise definition are the same across different systems and different configurations of the same system. If the units of a metric are not consistent, it is impossible to use the metric to compare the performances of the different systems. While the necessity for this characteristic would also seem obvious, it is not satisfied by many popular metrics, such as MIPS (Section 2.3.2) and MFLOPS (Section 2.3.3).
6. **Independence.** Many purchasers of computer systems decide which system to buy by comparing the values of some commonly used performance metric. As a result, there is a great deal of pressure on manufacturers to design their machines to optimize the value obtained for that particular metric, and to influence the composition of the metric to their benefit. To prevent corruption of its meaning, a good metric should be *independent* of such outside influences.

---

## 2.3 Processor and system performance metrics

---

A wide variety of performance metrics has been proposed and used in the computer field. Unfortunately, many of these metrics are not good in the sense defined above, or they are often used and interpreted incorrectly. The following subsections describe many of these common metrics and evaluate them against the above characteristics of a good performance metric.

### 2.3.1 The clock rate

In many advertisements for computer systems, the most prominent indication of performance is often the frequency of the processor's central clock. The implication to the buyer is that a 250 MHz system must always be faster at solving the user's problem than a 200 MHz system, for instance. However, this performance metric completely ignores how much computation is actually accomplished in each clock cycle, it ignores the complex interactions of the processor with the memory subsystem and the input/output subsystem, and it ignores the not at all unlikely fact that the processor may not be the performance bottleneck.

Evaluating the clock rate against the characteristics for a good performance metric, we find that it is very repeatable (characteristic 3) since it is a constant for a given system, it is easy to measure (characteristic 4) since it is most likely stamped on the box, the value of MHz is precisely defined across all systems so that it is consistent (characteristic 5), and it is independent of any sort of

manufacturers' games (characteristic 6). However, the unavoidable shortcomings of using this value as a performance metric are that it is nonlinear (characteristic 1), and unreliable (characteristic 2). As many owners of personal computer systems can attest, buying a system with a faster clock in no way assures that their programs will run correspondingly faster. Thus, we conclude that the processor's clock rate is not a good metric of performance.

### 2.3.2 MIPS

A *throughput* or *execution-rate* performance metric is a measure of the amount of computation performed per unit time. Since rate metrics are normalized to a common basis, such as seconds, they are very useful for comparing relative speeds. For instance, a vehicle that travels at  $50 \text{ m s}^{-1}$  will obviously traverse more ground in a fixed time interval than will a vehicle traveling at  $35 \text{ m s}^{-1}$ .

The MIPS metric is an attempt to develop a rate metric for computer systems that allows a direct comparison of their speeds. While in the physical world speed is measured as the distance traveled per unit time, MIPS defines the computer system's unit of 'distance' as the execution of an instruction. Thus, MIPS, which is an acronym for *millions of instructions executed per second*, is defined to be

$$\text{MIPS} = \frac{n}{t_e \times 10^6} \quad (2.1)$$

where  $t_e$  is the time required to execute  $n$  total instructions.

Defining the unit of 'distance' in this way makes MIPS easy to measure (characteristic 4), repeatable (characteristic 3), and independent (characteristic 6). Unfortunately, it does not satisfy any of the other characteristics of a good performance metric. It is not linear since, like the clock rate, a doubling of the MIPS rate does not necessarily cause a doubling of the resulting performance. It also is neither reliable nor consistent since it really does not correlate well to performance at all.

The problem with MIPS as a performance metric is that different processors can do substantially different amounts of computation with a single instruction. For instance, one processor may have a branch instruction that branches after checking the state of a specified condition code bit. Another processor, on the other hand, may have a branch instruction that first decrements a specified count register, and then branches after comparing the resulting value in the register with zero. In the first case, a single instruction does one simple operation, whereas in the second case, one instruction actually performs several operations. The failing of the MIPS metric is that each instruction corresponds to one unit of 'distance,' even though in this example the second instruction actually performs more real computation. These differences in the amount of computation per-

formed by an instruction are at the heart of the differences between RISC and CISC processors and render MIPS essentially useless as a performance metric. Another derisive explanation of the MIPS acronym is *meaningless indicator of performance* since it is really no better a measure of overall performance than is the processor's clock frequency.

### 2.3.3 MFLOPS

The MFLOPS performance metric tries to correct the primary shortcoming of the MIPS metric by more precisely defining the unit of 'distance' traveled by a computer system when executing a program. MFLOPS, which is an acronym for *millions of floating-point operations executed per second*, defines an arithmetic operation on two floating-point (i.e. fractional) quantities to be the basic unit of 'distance.' MFLOPS is thus calculated as

$$MFLOPS = \frac{f}{t_e \times 10^6} \quad (2.2)$$

where  $f$  is the number of floating-point operations executed in  $t_e$  seconds. The MFLOPS metric is a definite improvement over the MIPS metric since the results of a floating-point computation are more clearly comparable across computer systems than is the execution of a single instruction. An important problem with this metric, however, is that the MFLOPS rating for a system executing a program that performs no floating-point calculations is exactly zero. This program may actually be performing very useful operations, though, such as searching a database or sorting a large set of records.

A more subtle problem with MFLOPS is agreeing on exactly how to count the number of floating-point operations in a program. For instance, many of the Cray vector computer systems performed a floating-point division operation using successive approximations involving the reciprocal of the denominator and several multiplications. Similarly, some processors can calculate transcendental functions, such as sin, cos, and log, in a single instruction, while others require several multiplications, additions, and table look-ups. Should these operations be counted as a single floating-point operation or multiple floating-point operations? The first method would intuitively seem to make the most sense. The second method, however, would increase the value of  $f$  in the above calculation of the MFLOPS rating, thereby artificially inflating its value. This flexibility in counting the total number of floating-point operations causes MFLOPS to violate characteristic 6 of a good performance metric. It is also unreliable (characteristic 2) and inconsistent (characteristic 5).

**2.3.4    SPEC**

To standardize the definition of the actual result produced by a computer system in ‘typical’ usage, several computer manufacturers banded together to form the System Performance Evaluation Cooperative (SPEC). This group identified a set of integer and floating-point benchmark programs that was intended to reflect the way most workstation-class computer systems were actually used. Additionally, and, perhaps, most importantly, they also standardized the methodology for measuring and reporting the performance obtained when executing these programs.

The methodology defined consists of the following key steps.

1. Measure the time required to execute each program in the set on the system being tested.
2. Divide the time measured for each program in the first step by the time required to execute each program on a standard basis machine to normalize the execution times.
3. Average together all of these normalized values using the geometric mean (see Section 3.3.4) to produce a single-number performance metric.

While the SPEC methodology is certainly more rigorous than is using MIPS or MFLOPS as a measure of performance, it still produces a problematic performance metric. One shortcoming is that averaging together the individual normalized results with the geometric mean produces a metric that is not linearly related to a program’s actual execution time. Thus, the SPEC metric is not intuitive (characteristic 1). Furthermore, and more importantly, it has been shown to be an unreliable metric (characteristic 2) in that a given program may execute faster on a system that has a lower SPEC rating than it does on a competing system with a higher rating.

Finally, although the defined methodology appears to make the metric independent of outside influences (characteristic 6), it is actually subject to a wide range of tinkering. For example, many compiler developers have used these benchmarks as practice programs, thereby tuning their optimizations to the characteristics of this collection of applications. As a result, the execution times of the collection of programs in the SPEC suite can be quite sensitive to the particular selection of optimization flags chosen when the program is compiled. Also, the selection of specific programs that comprise the SPEC suite is determined by a committee of representatives from the manufacturers within the cooperative. This committee is subject to numerous outside pressures since each manufacturer has a strong interest in advocating application programs that will perform well on their machines. Thus, while SPEC is a significant step in the right direction towards defining a good performance metric, it still falls short of the goal.

### 2.3.5 QUIPS

The QUIPS metric, which was developed in conjunction with the HINT benchmark program, is a fundamentally different type of performance metric. (The details of the HINT benchmark and the precise definition of QUIPS are given in Section 7.2.3). Instead of defining the *effort* expended to reach a certain result as the measure of what is accomplished, the QUIPS metric defines the *quality of the solution* as a more meaningful indication of a user's final goal. The quality is rigorously defined on the basis of mathematical characteristics of the problem being solved. Dividing this measure of solution quality by the time required to achieve that level of quality produces QUIPS, or *quality improvements per second*.

This new performance metric has several of the characteristics of a good performance metric. The mathematically precise definition of 'quality' for the defined problem makes this metric insensitive to outside influences (characteristic 6) and makes it entirely self-consistent when it is ported to different machines (characteristic 5). It is also easily repeatable (characteristic 3) and it is linear (characteristic 1) since, for the particular problem chosen for the HINT benchmark, the resulting measure of quality is linearly related to the time required to obtain the solution.

Given the positive aspects of this metric, it still does present a few potential difficulties when used as a general-purpose performance metric. The primary potential difficulty is that it need not always be a reliable metric (characteristic 2) due to its narrow focus on floating-point and memory system performance. It is generally a very good metric for predicting how a computer system will perform when executing numerical programs. However, it does not exercise some aspects of a system that are important when executing other types of application programs, such as the input/output subsystem, the instruction cache, and the operating system's ability to multiprogram, for instance. Furthermore, while the developers have done an excellent job of making the HINT benchmark easy to measure (characteristic 4) and portable to other machines, it is difficult to change the quality definition. A new problem must be developed to focus on other aspects of a system's performance since the definition of quality is tightly coupled to the problem being solved. Developing a new problem to more broadly exercise the system could be a difficult task since it must maintain all of the characteristics described above.

Despite these difficulties, QUIPS is an important new type of metric that rigorously defines interesting aspects of performance while providing enough flexibility to allow new and unusual system architectures to demonstrate their capabilities. While it is not a completely general-purpose metric, it should prove to be very useful in measuring a system's numerical processing capabilities.

It also should be a strong stimulus for greater rigor in defining future performance metrics.

### 2.3.6 Execution time

Since we are ultimately interested in how quickly a given program is executed, the fundamental performance metric of any computer system is the time required to execute a given application program. Quite simply, the system that produces the smallest total execution time for a given application program has the highest performance. We can compare times directly, or use them to derive appropriate rates. However, without a precise and accurate measure of time, it is impossible to analyze or compare most any system performance characteristics. Consequently, it is important to know how to measure the execution time of a program, or a portion of a program, and to understand the limitations of the measuring tool.

The basic technique for measuring time in a computer system is analogous to using a stopwatch to measure the time required to perform some event. Unlike a stopwatch that begins measuring time from 0, however, a computer system typically has an internal counter that simply counts the number of clock ticks that have occurred since the system was first turned on. (See also Section 6.2.) A time interval then is measured by reading the value of the counter at the start of the event to be timed and again at the end of the event. The elapsed time is the difference between the two count values multiplied by the period of the clock ticks.

As an example, consider the program example shown in Figure 2.1. In this example, the `init_timer()` function initializes the data structures used to access the system's timer. This timer is a simple counter that is incremented continuously by a clock with a period defined in the variable `clock_cycle`. Reading the address pointed to by the variable `read_count` returns the current count value of the timer.

To begin timing a portion of a program, the current value in the timer is read and stored in `start_count`. At the end of the portion of the program being timed, the timer value is again read and stored in `end_count`. The difference between these two values is the total number of clock ticks that occurred during the execution of the event being measured. The total time required to execute this event is this number of clock ticks multiplied by the period of each tick, which is stored in the constant `clock_cycle`.

This technique for measuring the elapsed execution time of any selected portion of a program is often referred to as the *wall clock* time since it measures the total time that a user would have to wait to obtain the results produced by the program. That is, the measurement includes the time spent waiting for input/

```
main()
{
    int i;
    float a;

    init_timer();

    /* Read the starting time. */
    start_count = read_count;

    /* Stuff to be measured */
    for (i=0;i< 1000;i++) {
        a = i * a / 10;
    }

    /* Read the ending time. */
    end_count = read_count;

    elapsed_time = (end_count - start_count) * clock_cycle;
}
```

Figure 2.1. An example program showing how to measure the execution time of a portion of a program.

output operations to complete, memory paging, and other system operations performed on behalf of this application, all of which are integral components of the program's execution. However, when the system being measured is time-shared so that it is not dedicated to the execution of this one application program, this elapsed execution time also includes the time the application spends waiting while other users' applications execute.

Many researchers have argued that including this time-sharing overhead in the program's total execution time is unfair. Instead, they advocate measuring performance using the total time the processor actually spends executing the program, called the total *CPU time*. This time does not include the time the program is context switched-out while another application runs. Unfortunately, using only this CPU time as the performance metric ignores the waiting time that is inherent to the application as well as the time spent waiting on other programs. A better solution is to report both the CPU time and the total execution time so the reader can determine the significance of the time-sharing interference. The point is to be explicit about what information you are actually reporting to allow the reader to decide for themselves how believable your results are.

In addition to system-overhead effects, the measured execution time of an application program can vary significantly from one run to another since the program must contend with random events, such as the execution of background operating system tasks, different virtual-to-physical page mappings and cache mappings from explicitly random replacement policies, variable system load in a time-shared system, and so forth. As a result, a program's execution time is nondeterministic. It is important, then, to measure a program's total elapsed execution time several times and report at least the mean and variance of the times. Errors in measurements, along with appropriate statistical techniques to quantify them, are discussed in more detail in Chapter 4.

When it is measured as described above, the elapsed (wall clock) time measurement produces a performance metric that is intuitive, reliable, repeatable, easy to measure, consistent across systems, and independent of outside influences. Thus, since it satisfies all of the characteristics of a good performance metric, program execution time is one of the best metrics to use when analyzing computer system performance.

---

## 2.4 Other types of performance metrics

---

In addition to the processor-centric metrics described above, there are many other metrics that are commonly used in performance analysis. For instance, the system *response time* is the amount of time that elapses from when a user submits a request until the result is returned from the system. This metric is often used in analyzing the performance of online transaction-processing systems, for example. System *throughput* is a measure of the number of jobs or operations that are completed per unit time. The performance of a real-time video-processing system, for instance, may be measured in terms of the number of video frames that can be processed per second. The *bandwidth* of a communication network is a throughput measure that quantifies the number of bits that can be transmitted across the network per second. Many other *ad hoc* performance metrics are defined by performance analysts to suit the specific needs of the problem or system being studied.

---

## 2.5 Speedup and relative change

---

*Speedup* and *relative change* are useful metrics for comparing systems since they normalize performance to a common basis. Although these metrics are defined in terms of throughput or speed metrics, they are often calculated directly from execution times, as described below.

**Speedup.** The *speedup* of system 2 with respect to system 1 is defined to be a value  $S_{2,1}$  such that  $R_2 = S_{2,1}R_1$ , where  $R_1$  and  $R_2$  are the ‘speed’ metrics being compared. Thus, we can say that system 2 is  $S_{2,1}$  times faster than system 1. Since a speed metric is really a rate metric (i.e. throughput),  $R_1 = D_1/T_1$ , where  $D_1$  is analogous to the ‘distance traveled’ in time  $T_1$  by the application program when executing on system 1. Similarly,  $R_2 = D_2/T_2$ . Assuming that the ‘distance traveled’ by each system is the same,  $D_1 = D_2 = D$ , giving the following definition for speedup:

$$\text{Speedup of system 2 w.r.t. system 1} = S_{2,1} = \frac{R_2}{R_1} = \frac{D/T_2}{D/T_1} = \frac{T_1}{T_2}. \quad (2.3)$$

If system 2 is faster than system 1, then  $T_2 < T_1$  and the speedup ratio will be larger than 1. If system 2 is slower than system 1, however, the speedup ratio will be less than 1. This situation is often referred to as a *slowdown* instead of a speedup.

**Relative change.** Another technique for normalizing performance is to express the performance of a system as a percent change *relative* to the performance of another system. We again use the throughput metrics  $R_1$  and  $R_2$  as measures of the speeds of systems 1 and 2, respectively. The relative change of system 2 with respect to system 1, denoted  $\Delta_{2,1}$ , (that is, using system 1 as the basis) is then defined to be

$$\text{Relative change of system 2 w.r.t. system 1} = \Delta_{2,1} = \frac{R_2 - R_1}{R_1}. \quad (2.4)$$

Again assuming that the execution time of each system is measured when executing the same program, the ‘distance traveled’ by each system is the same so that  $R_1 = D/T_1$  and  $R_2 = D/T_2$ . Thus,

$$\Delta_{2,1} = \frac{R_2 - R_1}{R_1} = \frac{D/T_2 - D/T_1}{D/T_1} = \frac{T_1 - T_2}{T_2} = S_{2,1} - 1. \quad (2.5)$$

Typically, the value of  $\Delta_{2,1}$  is multiplied by 100 to express the relative change as a percentage with respect to a given basis system. This definition of relative change will produce a positive value if system 2 is faster than system 1, whereas a negative value indicates that the basis system is faster.

**Example.** As an example of how to apply these two different normalization techniques, the speedup and relative change of the systems shown in Table 2.1 are found using system 1 as the basis. From the raw execution times, we can easily see that system 4 is the fastest, followed by systems 2, 1, and 3, in that order. However, the speedup values give us a more precise indication of exactly how much faster one system is than the others. For instance, system 2 has a

**Table 2.1.** An example of calculating speedup and relative change using system 1 as the basis

System <i>x</i>	Execution time $T_x$ (s)	Speedup $S_{x,1}$	Relative change $\Delta_{x,1}$ (%)
1	480	1	0
2	360	1.33	+ 33
3	540	0.89	- 11
4	210	2.29	+ 129

speedup of 1.33 compared with system 1 or, equivalently, it is 33% faster. System 4 has a speedup ratio of 2.29 compared with system 1 (or it is 129% faster). We also see that system 3 is actually 11% slower than system 1, giving it a slowdown factor of 0.89.  $\diamond$

## 2.6 Means versus ends metrics

One of the most important characteristics of a performance metric is that it be reliable (characteristic 2). One of the problems with many of the metrics discussed above that makes them unreliable is that they measure what was done *whether or not it was useful*. What makes a performance metric reliable, however, is that it accurately and consistently measures *progress towards a goal*. Metrics that measure what was done, useful or not, have been called *means-based* metrics whereas *ends-based* metrics measure what is actually accomplished.

To obtain a feel for the difference between these two types of metrics, consider the vector dot-product routine shown in Figure 2.2. This program executes  $N$  floating-point addition and multiplication operations for a total of  $2N$  floating-point operations. If the time required to execute one addition is  $t_+$  cycles and one multiplication requires  $t_*$  cycles, the total time required to execute this program is  $t_1 = N(t_+ + t_*)$  cycles. The resulting execution rate then is

```
s = 0;
for (i = 1; i < N; i++)
    s = s + x[i] * y[i];
```

Figure 2.2. A vector dot-product example program.

$$R_1 = \frac{2N}{N(t_+ + t_*)} = \frac{2}{t_+ + t_*} \text{FLOPS/cycle.} \quad (2.6)$$

Since there is no need to perform the addition or multiplication operations for elements whose value is zero, it may be possible to reduce the total execution time if many elements of the two vectors are zero. Figure 2.3 shows the example from Figure 2.2 modified to perform the floating-point operations only for those nonzero elements. If the conditional *if* statement requires  $t_{\text{if}}$  cycles to execute, the total time required to execute this program is  $t_2 = N[t_{\text{if}} + f(t_+ + t_*)]$  cycles, where  $f$  is the fraction of  $N$  for which both  $x[i]$  and  $y[i]$  are nonzero. Since the total number of additions and multiplications executed in this case is  $2Nf$ , the execution rate for this program is

$$R_2 = \frac{2Nf}{N[t_{\text{if}} + f(t_+ + t_*)]} = \frac{2f}{t_{\text{if}} + f(t_+ + t_*)} \text{FLOPS/cycle.} \quad (2.7)$$

If  $t_{\text{if}}$  is four cycles,  $t_+$  is five cycles,  $t_*$  is ten cycles,  $f$  is 10%, and the processor's clock rate is 250 MHz (i.e. one cycle is 4 ns), then  $t_1 = 60N$  ns and  $t_2 = N[4 + 0.1(5 + 10)] \times 4$  ns =  $22N$  ns. The speedup of program 2 relative to program 1 then is found to be  $S_{2,1} = 60N/22N = 2.73$ .

Calculating the execution rates realized by each program with these assumptions produces  $R_1 = 2/(60 \text{ ns}) = 33 \text{ MFLOPS}$  and  $R_2 = 2(0.1)/(22 \text{ ns}) = 9.09 \text{ MFLOPS}$ . Thus, even though we have reduced the total execution time from  $t_1 = 60N$  ns to  $t_2 = 22N$  ns, the means-based metric (MFLOPS) shows that program 2 is 72% slower than program 1. The ends-based metric (execution time), however, shows that program 2 is actually 173% faster than program 1. We reach completely different conclusions when using these two different types of metrics because the means-based metric unfairly gives program 1 credit for all of the useless operations of multiplying and adding zero. This example highlights the danger of using the wrong metric to reach a conclusion about computer-system performance.

```
s = 0;
for (i = 1; i < N; i++)
    if (x[i] != 0 && y[i] != 0)
        s = s + x[i] * y[i];
```

Figure 2.3. The vector dot-product example program of Figure 2.2 modified to calculate only nonzero elements.

## 2.7 Summary

---

Fundamental to measuring computer-systems performance is defining an appropriate metric. This chapter identified several characteristics or criteria that are important for a ‘good’ metric of performance. Several common performance metrics were then introduced and analyzed in the context of these criteria. The definitions of speedup and relative change were also introduced. Finally, the concepts of ends-based versus means-based metrics were presented to clarify what actually causes a metric to be useful in capturing the actual performance of a computer system.

## 2.8 For further reading

---

- The following paper argues strongly for total execution time as the best measure of performance:

James E. Smith, ‘Characterizing Computer Performance with a Single Number,’ *Communications of the ACM*, October 1988, pp. 1202–1206.

- The QUIPS metric is described in detail in the following paper, which also introduced the idea of means-based versus ends-based metrics:

J. L. Gustafson and Q. O. Snell, ‘HINT: A New Way to Measure Computer Performance,’ *Hawaii International Conference on System Sciences*, 1995, pp. II:392–401.

- Some of the characteristics of the SPEC metric are discussed in the following papers:

Ran Giladi and Niv Ahituv, ‘SPEC as a Performance Evaluation Measure,’ *IEEE Computer*, Vol. 28, No. 8, August 1995, pp. 33–42.

Nikki Mirghafori, Margret Jacoby, and David Patterson, ‘Truth in SPEC Benchmarks,’ *ACM Computer Architecture News*, Vol. 23, No. 5, December 1995, pp. 34–42.

- Parallel computing systems are becoming more common. They present some interesting performance measurement problems, though, as discussed in

Lawrence A. Crowl, ‘How to Measure, Present, and Compare Parallel Performance,’ *IEEE Parallel and Distributed Technology*, Spring 1994, pp. 9–25.

## 2.9 Exercises

---

1. (a) Write a simple benchmark program to estimate the maximum effective MIPS rating of a computer system. Use your program to rank the performance of three different, but roughly comparable, computer systems.  
(b) Repeat part (a) using the maximum effective MFLOPS rating as the metric of performance.  
(c) Compare the rankings obtained in parts (a) and (b) with the ranking obtained by comparing the clock frequencies of the different systems.  
(d) Finally, compare your rankings with those published by authors using some standard benchmark programs, such as those available on the SPEC website.
2. What makes a performance metric ‘reliable’?
3. Classify each of the following metrics as being either means-based or ends-based; MIPS, MFLOPS, execution time, bytes of available memory, quality of a final answer, arithmetic precision, system cost, speedup, and reliability of an answer.
4. Devise an experiment to determine the following performance metrics for a computer system.
  - (a) The effective memory bandwidth between the processor and the data cache if all memory references are cache hits.
  - (b) The effective memory bandwidth if all memory references are cache misses.
5. What are the key differences between ‘wall clock time’ and ‘CPU time?’ Under what conditions should each one be used? Is it possible for these two different times to be the same?
6. The execution time required to read the current time from an interval counter is a minimum of at least one memory-read operation to obtain the current time value and one memory-write operation to store the value for later use. In some cases, it may additionally include a subroutine call and return operation. How does this timer ‘overhead’ affect the time measured when using such an interval timer to determine the duration of some event, such as the total execution time of a program?
7. Calculate the speedup and relative change of the four systems shown in Table 2.1 when using System 4 as the basis. How do your newly calculated values affect the relative rankings of the four systems?

# 6 Measurement tools and techniques

'When the only tool you have is a hammer, every problem begins to resemble a nail.'

*Abraham Maslow*

The previous chapters have discussed what performance metrics may be useful for the performance analyst, how to summarize measured data, and how to understand and quantify the systematic and random errors that affect our measurements. Now that we know what to do with our measured values, this chapter presents several tools and techniques for actually measuring the values we desire.

The focus of this chapter is on fundamental measurement concepts. The goal is not to teach you how to use specific measurement tools, but, rather, to help you understand the strengths and limitations of the various measurement techniques. By the end of this chapter, you should be able to select an appropriate measurement technique to determine the value of a desired performance metric. You also should have developed some understanding of the trade-offs involved in using the various types of tools and techniques.

## 6.1 Events and measurement strategies

There are many different types of performance metrics that we may wish to measure. The different strategies for measuring the values of these metrics are typically based around the idea of an *event*, where an event is some predefined change in the system state. The precise definition of a specific event is up to the performance analyst and depends on the metric being measured. For instance, an event may be defined to be a memory reference, a disk access, a network communication operation, a change in a processor's internal state, or some pattern or combination of other subevents.

**6.1.1 Events-type classification**

The different types of metrics that a performance analyst may wish to measure can be classified into the following categories based on the type of event or events that comprise the metric.

1. **Event-count metrics.** Metrics that fall into this category are those that are simple counts of the number of times a specific event occurs. Examples of event-count metrics include the number of page faults in a system with virtual memory, and the number of disk input/output requests made by a program.
2. **Secondary-event metrics.** These types of metrics record the values of some secondary parameters whenever a given event occurs. For instance, to determine the average number of messages queued in the send buffer of a communication port, we would need to record the number of messages in the queue each time a message was added to, or removed from, the queue. Thus, the triggering event is a message-enqueue or -dequeue operation, and the metrics being recorded are the number of messages in the queue and the total number of queue operations. We may also wish to record the size (e.g. the number of bytes) of each message sent to later determine the average message size.
3. **Profiles.** A profile is an aggregate metric used to characterize the overall behavior of an application program or of an entire system. Typically, it is used to identify where the program or system is spending its execution time.

**6.1.2 Measurement strategies**

The above event-type classification can be useful in helping the performance analyst decide on a specific strategy for measuring the desired metric, since different types of measurement tools are appropriate for measuring different types of events. These different measurement tools can be categorized on the basis of the fundamental strategy used to determine the actual values of the metrics being measured. One important concern with any measurement strategy is how much it *perturbs* the system being measured. This aspect of performance measurement is discussed further in Section 6.6.

1. **Event-driven.** An event-driven measurement strategy records the information necessary to calculate the performance metric whenever the preselected event or events occur. The simplest type of event-driven measurement tool uses a simple counter to directly count the number of occurrences of a specific event. For example, the desired metric may be the number of page faults that occur during the execution of an application program. To find this value, the performance analyst most likely would have to modify the page-fault-handling

routine in the operating system to increment a counter whenever the routine is entered. At the termination of the program's execution, an additional mechanism must be provided to dump the contents of the counter.

One of the advantages of an event-driven strategy is that the system overhead required to record the necessary information is incurred only when the event of interest actually occurs. If the event never occurs, or occurs only infrequently, the perturbation to the system will be relatively small. This characteristic can also be a disadvantage, however, when the events being monitored occur very frequently.

When recording high-frequency events, a great deal of overhead may be introduced into a program's execution, which can significantly alter the program's execution behavior compared with its uninstrumented execution. As a result, what the measurement tool measures need not reflect the typical or average behavior of the system. Furthermore, the time between measurements depends entirely on when the measured events occur so that the inter-event time can be highly variable and completely unpredictable. This can increase the difficulty of determining how much the measurement tool actually perturbs the executing program. Event-driven measurement tools are usually considered most appropriate for low-frequency events.

2. **Tracing.** A tracing strategy is similar to an event-driven strategy, except that, rather than simply recording that fact that the event has occurred, some portion of the system state is recorded to uniquely identify the event. For example, instead of simply counting the number of page faults, a tracing strategy may record the addresses that caused each of the page faults. This strategy obviously requires significantly more storage than would a simple count of events. Additionally, the time required to save the desired state, either by storing it within the system's memory or by writing to a disk, for instance, can significantly alter the execution of the program being measured.
3. **Sampling.** In contrast to an event-driven measurement strategy, a sampling strategy records at fixed time intervals the portion of the system state necessary to determine the metric of interest. As a result, the overhead due to this strategy is independent of the number of times a specific event occurs. It is instead a function of the sampling frequency, which is determined by the resolution necessary to capture the events of interest.

The sampling of the state of the system occurs at fixed time intervals that are independent of the occurrence of specific events. Thus, not every occurrence of the events of interest will be recorded. Rather, a sampling strategy produces a statistical summary of the overall behavior of the system. Consequently, events that occur infrequently may be completely missed by this statistical approach. Furthermore, each run of a sampling-based experiment is likely to produce a different result since the samples occur asynchronously with respect

to a program's execution. Nevertheless, while the exact behavior may differ, the statistical behavior should remain approximately the same.

4. **Indirect.** An indirect measurement strategy must be used when the metric that is to be determined is not directly accessible. In this case, you must find another metric that can be measured directly, from which you then can deduce or derive the desired performance metric. Developing an appropriate indirect measurement strategy, and minimizing its overhead, relies almost completely on the cleverness and creativity of the performance analyst.

The unique characteristics of these measurement strategies make them more or less appropriate for different situations. Program tracing can provide the most detailed information about the system being monitored. An event-driven measurement tool, on the other hand, typically provides only a higher-level summary of the system behavior, such as overall counts or average durations. The information supplied both by an event-driven measurement tool and by a tracing tool is exact, though, such as the precise number of times a certain subroutine is executed. In contrast, the information provided by a sampling strategy is statistical in nature. Thus, repeating the same experiment with an event-driven or tracing tool will produce the same results each time whereas the results produced with a sampling tool will vary slightly each time the experiment is performed.

The system resources consumed by the measurement tool itself as it collects data will strongly affect how much perturbation the tool will cause in the system. As mentioned above, the overhead of an event-driven measurement tool is directly proportional to the number of occurrences of the event being measured. Events that occur frequently may cause this type of tool to produce substantial perturbation as a byproduct of the measurement process. The overhead of a sampling-based tool, however, is independent of the number of times any specific event occurs. The perturbation caused by this type of tool is instead a function of the sampling interval, which can be controlled by the experimenter or the tool builder. A trace-based tool consumes the largest amount of system resources, requiring both processor resources (i.e. time) to record each event and potentially enormous amounts of storage resources to save each event in the trace. As a result, tracing tends to produce the largest system perturbation.

Each indirect measurement tool must be uniquely adapted to the particular aspect of the system performance it attempts to measure. Therefore, it is impossible to make any general statements about a measurement tool that makes use of an indirect strategy. The key to implementing a tool to measure a specific performance metric is to match the characteristics of the desired metric with the appropriate measurement strategy. Several of the fundamental techniques that have been used for implementing the various measurement strategies are described in the following sections.

## 6.2 Interval timers

One of the most fundamental measuring tools in computer-system performance analysis is the *interval timer*. An interval timer is used to measure the execution time of an entire program or any section of code within a program. It can also provide the time basis for a sampling measurement tool. Although interval timers are relatively straightforward to use, understanding how an interval timer is constructed helps the performance analyst determine the limitations inherent in this type of measurement tool.

Interval timers are based on the idea of counting the number of clock pulses that occur between two predefined events. These events are typically identified by inserting calls to a routine that reads the current timer count value into a program at the appropriate points, such as shown previously in the example in Figure 2.1. There are two common implementations of interval timers, one using a hardware counter, and the other based on a software interrupt.

**Hardware timers.** The hardware-based interval timer shown in Figure 6.1 simply counts the number of pulses it receives at its clock input from a free-running clock source. The counter is typically reset to 0 when the system is first powered up so that the value read from the counter is the number of clock ticks that have occurred since that time. This value is used within a program by reading the memory location that has been mapped to this counter by the manufacturer of the system.

Assume that the value read at the start of the interval being measured is  $x_1$  and the value read at the end of the interval is  $x_2$ . Then the total time that has elapsed between these two read operations is  $T_e = (x_2 - x_1)T_c$ , where  $T_c$  is the period of the clock input to the counter.

**Software timers.** The primary difference between a software-interrupt-based interval timer, shown in Figure 6.2, and a hardware-based timer is that the counter accessible to an application program in the software-based implementa-

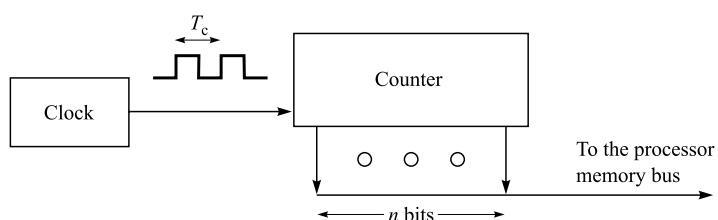


Figure 6.1 A hardware-based interval timer uses a free-running clock source to continuously increment an  $n$ -bit counter. This counter can be read directly by the operating system or by an application program. The period of the clock,  $T_c$ , determines the resolution of the timer.

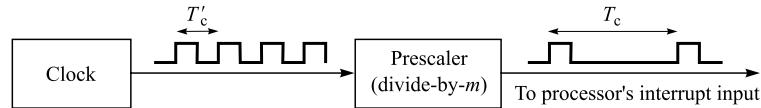


Figure 6.2 A software interrupt-based timer divides down a free-running clock to produce a processor interrupt with the period  $T_c$ . The interrupt service routine then maintains a counter variable in memory that it increments each time the interrupt occurs.

tion is not directly incremented by the free-running clock. Instead, the hardware clock is used to generate a processor interrupt at regular intervals. The interrupt-service routine then increments a counter variable it maintains, which is the value actually read by an application program. The value of this variable then is a count of the number of interrupts that have occurred since the count variable was last initialized. Some systems allow an application program to reset this counter. This feature allows the timer to always start from zero when timing the duration of an event.

The period of the interrupts in the software-based approach corresponds to the period of the timer. As before, we denote this period  $T_c$  so that the total time elapsed between two readings of the software counter value is again  $T_e = (x_2 - x_1)T_c$ . The processor interrupt is typically derived from a free-running clock source that is divided by  $m$  through a prescaling counter, as shown in Figure 6.2. This prescaler is necessary in order to reduce the frequency of the interrupt signal fed into the processor. Interrupts would occur much too often, and thus would generate a huge amount of processor overhead, if this prescaling were not done.

**Timer rollover.** One important consideration with these types of interval timers is the number of bits available for counting. This characteristic directly determines the longest interval that can be measured. (The complementary issue of the shortest interval that can be measured is discussed in Section 6.2.2.) A binary counter used in a hardware timer, or the equivalent count variable used in a software implementation, is said to ‘roll over’ to zero as its count undergoes a transition from its maximum value of  $2^n - 1$  to the zero value, where  $n$  is the number of bits in the counter.

If the counter rolls over between the reading of the counter at the start of the interval being measured and the reading of the counter at the end, the difference of the count values,  $x_2 - x_1$ , will be a negative number. This negative value is obviously not a valid measurement of the time interval. Any program that uses an interval timer must take care to ensure that this type of roll over can never occur, or it must detect and, possibly, correct the error. Note that a negative value that occurs due to a single roll over of the counter can be converted to the appropriate value by adding the maximum count value,  $2^n$ , to the negative value

obtained when subtracting  $x_1$  from  $x_2$ . Table 6.1 shows the maximum time between timer roll overs for various counter widths and input clock periods.

### 6.2.1 Timer overhead

The implementation of an interval timer on a specific system determines how the timer must be used. In general, though, we can think of using an interval timer to measure any portion of a program, much as we would use a stopwatch to time a runner on a track, for instance. In particular, we typically would use an interval time within a program as follows:

```
x_start = read_timer();
<event being timed>
x_end = read_timer();
elapsed_time = (x_end - x_start) * t_cycle;
```

When it is used in this way, we can see that the time we actually measure includes more than the time required by the event itself. Specifically, accessing the timer requires a minimum of one memory-read operation. In some implementations, reading the timer may require as much as a call to the operating-system kernel, which can be very time-consuming. Additionally, the value read from the timer must be stored somewhere before the event being timed begins. This requires at least one store operation, and, in some systems, it could require substantially more. These operations must be performed twice, once at the start of the event, and once again at the end. Taken altogether, these operations can add up to a significant amount of time relative to the duration of the event itself.

To obtain a better understanding of this timer overhead, consider the time line shown in Figure 6.3. Here,  $T_1$  is the time required to read the value of the interval timer's counter. It may be as short as a single memory read, or as long as a call into the operating-system kernel. Next,  $T_2$  is the time required to store the current time. This time includes any time in the kernel after the counter has been read, which would include, at a minimum, the execution of the return instruction. Time  $T_3$  is the actual duration of the event we are trying to measure. Finally, the time from when the event ends until the program actually reads the counter value again is  $T_4$ . Note that reading the counter this second time involves the same set of operations as the first read of the counter so that  $T_4 = T_1$ .

Assigning these times to each of the components in the timing operation now allows us to compare the timer overhead with the time of the event itself, which is what we actually want to know. This event time,  $T_e$  is time  $T_3$  in our time line, so that  $T_e = T_3$ . What we measure, however, is  $T_m = T_2 + T_3 + T_4$ . Thus, our

**Table 6.1** The maximum time available before a binary interval timer with  $n$  bits and an input clock with a period of  $T_c$  rolls over is  $T_c 2^n$

$T_c$	Counter width, $n$				
	16	24	32	48	64
10 ns	655 $\mu$ s	168 ms	42.9 s	32.6 days	58.5 centuries
100 ns	6.55 ms	1.68 s	7.16 min	326 days	585 centuries
1 $\mu$ s	65.5 ms	16.8 s	1.19 h	9.15 years	5,850 centuries
10 $\mu$ s	655 ms	2.8 min	11.9 h	89.3 years	58,500 centuries
100 $\mu$ s	6.55 s	28.0 min	4.97 days	893 years	585,000 centuries
1 ms	1.09 min	4.66 h	49.7 days	89.3 centuries	5,850,000 centuries

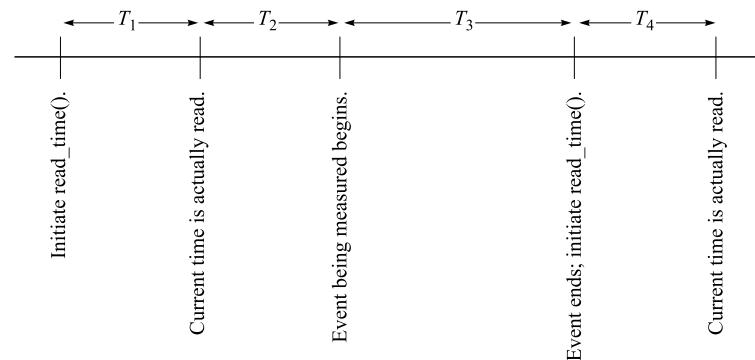


Figure 6.3 The overhead incurred when using an interval timer to measure the execution time of any portion of a program can be understood by breaking down the operations necessary to use the timer into the components shown here.

desired measurement is  $T_e = T_m - (T_2 + T_4) = T_m - (T_1 + T_2)$ , since  $T_4 = T_1$ . We call  $T_1 + T_2$  the *timer overhead* and denote it  $T_{\text{ovhd}}$ .

If the interval being measured is substantially larger than the timer overhead, then the timer overhead can simply be ignored. If this condition is not satisfied, though, then the timer overhead should be carefully measured and subtracted from the measurement of the event under consideration. It is important to recognize, however, that variations in measurements of the timer overhead itself can often be quite large relative to variations in the times measured for the event. As a result, measurements of intervals whose duration is of the same order of magnitude as the timer overhead should be treated with great suspicion. A good rule of thumb is that the event duration,  $T_e$ , should be 100–1,000 times larger than the timer overhead,  $T_{\text{ovhd}}$ .

### 6.2.2 Quantization errors

The smallest change that can be detected and displayed by an interval timer is its *resolution*. This resolution is a single clock tick, which, in terms of time, is the period of the timer's clock input,  $T_c$ . This finite resolution introduces a random *quantization error* into all measurements made using the timer.

For instance, consider an event whose duration is  $n$  ticks of the clock input, plus a little bit more. That is,  $T_e = nT_c + \Delta$ , where  $n$  is a positive integer and  $0 < \Delta < T_c$ . If, when one is measuring this event, the timer value is read shortly after the event has actually begun, as shown in Figure 6.4(a), the timer will count  $n$  clock ticks before the end of the event. The total execution time reported then will be  $nT_c$ . If, on the other hand, there is slightly less time between the actual start of the event and the point at which the timer value is read, as shown in Figure 6.4(b), the timer will count  $n + 1$  clock ticks before the end of the event is detected. The total time reported in this case will then be  $(n + 1)T_c$ .

In general, the actual event time is within the range  $nT_c < T_e < (n + 1)T_c$ . Thus, the fact that events are typically not exactly whole number factors of the timer's clock period causes the time value reported to be rounded either up or down by one clock period. This rounding is completely unpredictable and is one readily identifiable (albeit possibly small) source of random errors in our measurements (see Section 4.2). Looking at this quantization effect another way, if we made ten measurements of the same event, we would expect that approximately five of them would be reported as  $nT_c$  with the remainder reported as  $(n + 1)T_c$ . If  $T_c$  is large relative to the event being measured, this quantization effect can make it impossible to directly measure the duration of the event. Consequently, we typically would like  $T_c$  to be as small as possible, within the constraints imposed by the number of bits available in the timer (see Table 6.1).

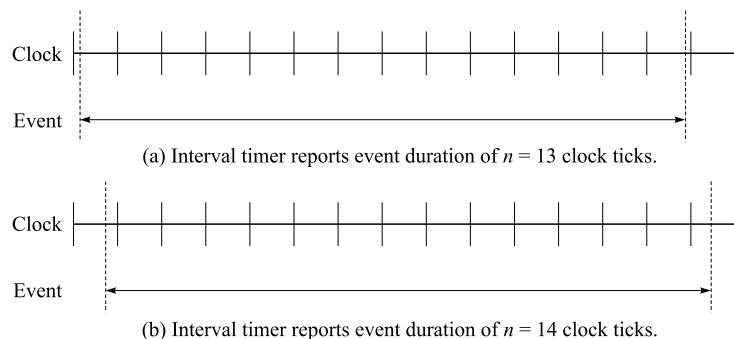


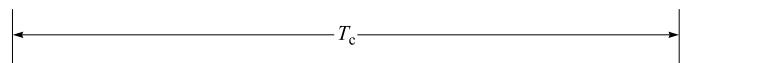
Figure 6.4 The finite resolution of an interval timer causes quantization of the reported duration of the events measured.

### 6.2.3 Statistical measures of short intervals

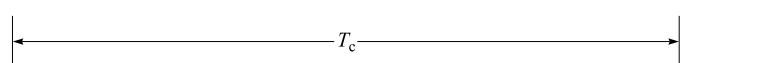
Owing to the above quantization effect, we cannot directly measure events whose durations are less than the resolution of the timer. Similarly, quantization makes it difficult to accurately measure events with durations that are only a few times larger than the timer's resolution. We can, however, make many measurements of a short duration event to obtain a statistical estimate of the event's duration.

Consider an event whose duration is smaller than the timer's resolution, that is,  $T_e < T_c$ . If we measure this interval once, there are two possible outcomes. If we happen to start our measurement such that the event straddles the active edge of the clock that drives the timer's internal counter, as shown in Figure 6.5(a), we will see the clock advance by one tick. On the other hand, since  $T_e < T_c$ , it is entirely possible that the event will begin and end within one clock period, as shown in Figure 6.5(b). In this case, the timer will not advance during this measurement. Thus, we have a Bernoulli experiment whose outcome is 1 with probability  $p$ , which corresponds to the timer advancing by one tick while measuring the event. If the clock does not advance, though, the outcome is 0 with probability  $1 - p$ .

Repeating this measurement  $n$  times produces a distribution that approximates a binomial distribution. (It is only approximate since, for a true binomial dis-



(a) Event  $T_e$  straddles the active edge of the interval timer.



(b) Event  $T_e$  begins and ends within the resolution of the interval timer.

Figure 6.5 When one is measuring an event whose duration is less than the resolution of the interval timer, that is,  $T_e < T_c$ , there are two possible outcomes for each measurement. Either the event happens to straddle the active edge of the timer's clock, in which case the counter advances by one tick, or the event begins and completes between two clock edges. In the latter case, the interval timer will show the same count value both before and after the event. Measuring this event multiple times approximates a binomial distribution.

tribution, each of the  $n$  measurements must be independent. However, in a real system it is possible that obtaining an outcome of 0 in one measurement makes it more likely that one will obtain a 0 in the next measurement, for instance. Nevertheless, this approximation appears to work well in practice.) If the number of outcomes that produce 1 is  $m$ , then the ratio  $m/n$  should approximate the ratio of the duration of the event being measured to the clock period,  $T_e/T_c$ . Thus, we can estimate the average duration of this event to be

$$T_e = \frac{m}{n} T_c. \quad (6.1)$$

We can then use the technique for calculating a confidence interval for a proportion (see Section 4.4.3) to obtain a confidence interval for this average event time.<sup>1</sup>

**Example.** We wish to measure an event whose duration we suspect is less than the 40  $\mu\text{s}$  resolution of our interval timer. Out of  $n = 10,482$  measurements of this event, we find that the clock actually advances by one tick during  $m = 852$  of them. For a 95% confidence level, we construct the interval for the ratio  $m/n = 852/10,482$  as follows:

$$(c_1, c_2) = \frac{852}{10,482} \pm (1.96) \sqrt{\frac{\frac{852}{10,482} \left(1 - \frac{852}{10,482}\right)}{10,482}} = (0.0786, 0.0840). \quad (6.2)$$

Scaling this interval by the timer's clock period gives us the 95% confidence interval (3.14, 3.36) $\mu\text{s}$  for the duration of this event. ◇

---

### 6.3 Program profiling

---

A *profile* provides an overall view of the execution behavior of an application program. More specifically, it is a measurement of how much time, or the fraction of the total time, the system spends in certain states. A profile of a program can be useful for showing how much time the program spends executing each of its various subroutines, for instance. This type of information is often used by a programmer to identify those portions of the program that consume the largest fraction of the total execution time. Once the largest time consumers have been identified, they can, one assumes, be enhanced to thereby improve performance.

Similarly, when a profile of an entire system multitasking among several different applications is taken, it can be used by a system administrator to find system-level performance bottlenecks. This information can be used in turn to

<sup>1</sup> The basic idea behind this technique was first suggested by Peter H. Danzig and Steve Melvin in an unpublished technical report from the University of Southern California.

tune the performance of the overall system by adjusting such parameters as buffer sizes, time-sharing quanta, disk-access policies, and so forth.

There are two distinct techniques for creating a program profile – program-counter (PC) sampling and basic-block counting. Sampling can also be used to generate a profile of a complete system.

### 6.3.1 PC sampling

*Sampling* is a general statistical measurement technique in which a subset (i.e. a sample) of the members of a population being examined is selected at random. The information of interest is then gathered from this subset of the total population. It is assumed that, since the samples were chosen completely at random, the characteristics of the overall population will approximately follow the same proportions as do the characteristics of the subset actually measured. This assumption allows conclusions about the overall population to be drawn on the basis of the complete information obtained from a small subset of this population.

While this traditional population sampling selects all of the samples to be tested at (essentially) the same time, a slightly different approach is required when using sampling to generate a profile of an executing program. Instead of selecting all of the samples to be measured at once, samples of the executing program are taken at fixed points in time. Specifically, an external periodic signal is generated by the system that interrupts the program at fixed intervals. Whenever one of these interrupts is detected, appropriate state information is recorded by the interrupt-service routine.

For instance, when one is generating a profile for a single executing program, the interrupt-service routine examines the return-address stack to find the address of the instruction that was executing when the interrupt occurred. Using symbol-table information previously obtained from the compiler or assembler, this program-instruction address is mapped onto a specific subroutine identifier,  $i$ . The value  $i$  is used to index into a single-dimensional array,  $H$ , to then increment the element  $H_i$  by one. In this way, the interrupt-service routine generates a histogram of the number of times each subroutine in the program was being executed when the interrupt occurred.

The ratio  $H_i/n$  is the fraction of the program's total execution time that it spent executing in subroutine  $i$ , where  $n$  is the total number of interrupts that occurred during the program's execution. Multiplying the period of the interrupt by these ratios provides an estimate of the total time spent executing in each subroutine.

It is important to remember that sampling is a statistical process in which the characteristics of an entire population (in our present situation, the execution

behavior of an entire program or system) are inferred from a randomly selected subset of the overall population. The calculated values of these inferences are, therefore, subject to random errors. Not surprisingly, we can calculate a confidence interval for these proportions to obtain a feel for the precision of our sampling experiment.

**Example.** Suppose that we use a sampling tool that interrupts an executing program every  $T_c = 10$  ms. Including the time required to execute the interrupt-service routine, the program executes for a total of 8 s. If  $H_X = 12$  of the  $n = 800$  samples find the program counter somewhere in subroutine X when the interrupt occurred, what is the fraction of the total time the program spends executing this subroutine?

Since there are 800 samples in total, we conclude that the program spends 1.5% ( $12/800 = 0.015$ ) of its time in subroutine X. Using the procedure from Section 4.4.3, we calculate a 99% confidence interval for this proportion to be

$$(c_1, c_2) = 0.015 \pm 2.576 \sqrt{\frac{0.015(1 - 0.015)}{800}} = (0.0039, 0.0261). \quad (6.3)$$

So, with 99% confidence, we estimate that the program spends between 0.39% and 2.6% of its time executing subroutine X. Multiplying by the period of the interrupt, we estimate that, out of the 8 s the program was executing, there is a 99% chance that it spent between 31 ( $0.0039 \times 8$ ) and 210 ( $0.0261 \times 8$ ) ms executing subroutine X. ◇

The confidence interval calculated in the above example produces a rather large range of times that the program could be spending in subroutine X. Put in other terms, if we were to repeat this experiment several times, we would expect that, in 99% of the experiments, from three to 21 of the 800 samples would come from subroutine X. While this 7 : 1 range of possible execution times appears large, we estimate that subroutine X still accounts for less than 3% of the total execution time. Thus, we most likely would start our program-tuning efforts on a routine that consumes a much larger fraction of the total execution time.

This example does demonstrate the importance of having a sufficient number of samples in each state to produce reliable information, however. To reduce the size of the confidence interval in this example we need more samples of each event. Obtaining more samples per event requires either sampling for a longer period of time, or increasing the sampling rate. In some situations, we can simply let the program execute for a longer period of time. This will increase the total number of samples and, hence, the number of samples obtained for each subroutine.

Some programs have a fixed duration, however, and cannot be forced to execute for a longer period. In this situation, we can run the program multiple

times and simply add the samples from each run. The alternative of increasing the sampling frequency will not always be possible, since the interrupt period is often fixed by the system or the profiling tool itself. Furthermore, increasing the sampling frequency increases the number of times the interrupt-service routine is executed, which increases the perturbation to the program. Of course, each run of the program must be performed under identical conditions. Otherwise, if the test conditions are not identical, we are testing two essentially different systems. Consequently, in this case, the two sets of samples cannot be simply added together to form one larger sample set.

It is also important to note that this sampling procedure implicitly assumes that the interrupt occurs completely asynchronously with respect to any events in the program being profiled. Although the interrupts occur at fixed, predefined intervals, if the program events and the interrupt are asynchronous, the interrupts will occur at random points in the execution of the program being sampled. Thus, the samples taken at these points are completely independent of each other. This sample independence is critical to obtaining accurate results with this technique since any synchronism between the events in the program and the interrupt will cause some areas of the program to be sampled more often than they should, given their actual frequency of occurrence.

### 6.3.2 Basic-block counting

The sampling technique described above provides a statistical profile of the behavior of a program. An alternative approach is to produce an exact execution profile by counting the number of times each *basic block* is executed. A basic block is a sequence of processor instructions that has no branches into or out of the sequence, as shown in Figure 6.6. Thus, once the first instruction in a block begins executing, it is assured that all of the remaining instructions in the block will be executed. The instructions in a basic block can be thought of as a computation that will always be executed as a single unit.

A program's basic-block structure can be exploited to generate a profile by inserting into each basic block additional instructions. These additional instructions simply count the number of times the block is executed. When the program terminates, these values form a histogram of the frequency of the basic-block executions. Just like the histogram produced with sampling, this basic-block histogram shows which portions of the program are executed most frequently. In this case, though, the resolution of the information is at the basic-block level instead of the subroutine level. Since a basic block executes as an indivisible unit, complete instruction-execution-frequency counts can also be obtained from these basic-block counts.

```

1. $37:    la      $25, __iob
2.          lw      $15, 0($25)
3.          addu   $9, $15, -1
4.          sw      $9, 0($25)
5.          la      $8, __iob
6.          lw      $11, 0($8)
7.          bge   $11, 0, $38
8.          move   $4, $8
9.          jal     __filbuf
10.         move   $17, $2
11. $38:    la      $12, __iob
.
.
```

Figure 6.6 A basic block is a sequence of instructions with no branches into or out of the block. In this example, one basic block begins at statement 1 and ends at statement 7. A second basic block begins at statement 8 and ends at statement 9. Statement 10 is a basic block consisting of only one instruction. Statement 11 begins another basic block since it is the target of an instruction that branches to label \$38.

One of the key differences between this basic-block profile and a profile generated through sampling is that the basic-block profile shows the *exact* execution frequencies of all of the instructions executed by a program. The sampling profile, on the other hand, is only a statistical estimate of the frequencies. Hence, if a sampling experiment is run a second time, the precise execution frequencies will most likely be at least slightly different. A basic-block profile, however, will produce exactly the same frequencies whenever the program is executed with the same inputs.

Although the repeatability and exact frequencies of basic-block counting would seem to make it the obvious profiling choice over a sampling-based profile, modifying a program to count its basic-block executions can add a substantial amount of run-time overhead. For instance, to instrument a program for basic-block counting would require the addition of at least one instruction to increment the appropriate counter when the block begins executing to each basic block. Since the counters that need to be incremented must be unique for each basic block, it is likely that additional instructions to calculate the appropriate offset for the current block into the array of counters will be necessary.

In most programs, the number of instructions in a basic block is typically between three and 20. Thus, the number of instructions executed by the instrumented program is likely to increase by at least a few percent and possibly as much as 100% compared with the uninstrumented program. These additional instructions can substantially increase the total running time of the program.

Furthermore, the additional memory required to store the counter array, plus the execution of the additional instructions, can cause other substantial perturbations. For instance, these changes to the program can significantly alter its memory behavior.

So, while basic-block counting provides exact profile information, it does so at the expense of substantial overhead. Sampling, on the other hand, distributes its perturbations randomly throughout a program's execution. Also, the total perturbation due to sampling can be controlled somewhat by varying the period of the sampling interrupt interval. Nevertheless, basic-block counting can be a useful tool for precisely characterizing a program's execution profile. Many compilers, in fact, have compile-time flags a user can set to automatically insert appropriate code into a program as it is compiled to generate the desired basic-block counts when it is subsequently executed.

---

## 6.4 Event tracing

---

The information captured through a profiling tool provides a summary picture of the overall execution of a program. An often-useful type of information that is ignored in this type of profile summary, however, is the time-ordering of events. A basic-block-counting profile can show the type and frequency of each of the instructions executed, for instance, but it does not provide any information about the order in which the instructions were executed. When this sequencing information is important to the analysis being performed, a program trace is the appropriate choice.

A *trace* of a program is a dynamic list of the events generated by the program as it executes. The events that comprise a trace can be any events that you can find a way to monitor, such as a time-ordered list of all of the instructions executed by a program, the sequence of memory addresses accessed by a program, the sequence of disk blocks referenced by the file system, the sizes and destinations of all messages sent over a network, and so forth. The level of detail provided in a trace is entirely determined by the performance analyst's ability to gather the information necessary for the problem at hand.

Traces themselves can be analyzed to characterize the overall behavior of a program, much as a profile characterizes a program's behavior. However, traces are probably more typically used as the input to drive a simulator. For instance, traces of the memory addresses referenced by a program are often used to drive cache simulators. Similarly, traces of the messages sent by an application program over a communication network are often used to drive simulators for evaluating changes to communication protocols.

### 6.4.1 Trace generation

The overall tracing process is shown schematically in Figure 6.7. A tracing system typically consists of two main components. The first is the application being traced, which is the component that actually *generates* the trace. The second main component is the trace *consumer*. This is the program, such as a simulator, that actually uses the information being generated. In between the trace generator and the consumer is often a large disk file on which to store the trace. Storing the trace allows the consumer to be run many times against an unchanging trace to allow comparison experiments without the expense of regenerating the trace. Since the trace can be quite large, however, it will not always be possible or desirable to store the trace on an intermediate disk. In this case, it is possible to consume the trace *online* as it is generated.

A wide range of techniques have been developed for generating traces. Several of these approaches are summarized below.

1. **Source-code modification.** Perhaps the most straightforward approach for generating a program trace is to modify the source code of the program to be traced. For instance, the programmer may add additional tracing statements to the source code, as shown in Figure 6.8. When the program is subsequently compiled and executed, these additional program statements will be executed, thereby generating the desired trace. One advantage of this approach is that the programmer can trace only the desired events. This can help reduce the volume of trace data generated. One major disadvantage is that inserting trace points is typically a manual process and is, therefore, very time-consuming and prone to error.
2. **Software exceptions.** Some processors have been constructed with a mode that forces a software exception just before the execution of each instruction. The

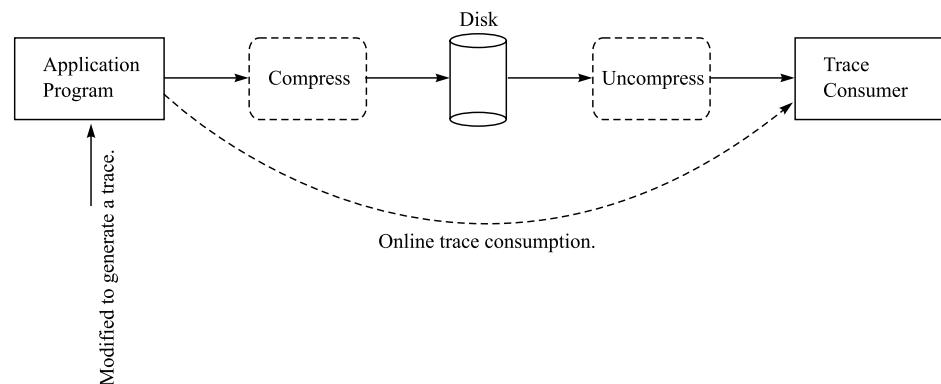


Figure 6.7 The overall process used to generate, store, and consume a program trace.

```

        sum_x = 0.0;
        trace(1);
        sum_xx = 0.0;
        trace(2);
        for (i = 1; i <= n; i++)
        trace(3);
        {
            sum_x += x[i];
            trace(4);
            sum_xx += (x[i]*x[i]);
            trace(5);
        }
        mean = sum_x / n;
        trace(6);
        var = ((n * sum_xx) - (sum_x * sum_x)) / (n * (n-1));
        trace(7);
        std_dev = sqrt(var);
        trace(8);
        z_p = unit_normal(1 - (0.5 * alpha));
        trace(9);
        half_int = z_p * std_dev / sqrt(n);
        trace(10);
        c1 = mean - half_int;
        trace(11);
        c2 = mean + half_int;
        trace(12);
    
```

(a) The original source program with calls to the tracing routine inserted.

```

trace(i)
{ print(i,time); }

```

(b) The trace routine simply prints the statement number, *i*, and the current time.

Figure 6.8 Program tracing can be performed by inserting additional statements into the source code to call a tracing subroutine at appropriate points.

exception-processing routine can decode the instruction to determine its operands. The instruction type, address, and operand addresses and values can then be stored for later use. This approach was implemented using the T-bit in Digital Equipment Corporation's VAX processor series and in the Motorola 68000 processor family. Executing with the trace mode enabled on these processors slowed down a program's execution by a factor of about 1,000.

3. **Emulation.** An emulator is a program that makes the system on which it executes appear to the outside world as if it were something completely different. For example, the Java Virtual Machine is a program that executes application programs written in the Java programming language by emulating the operation of a processor that implements the Java byte-code instruction set. This emulation obviously slows down the execution of the application program compared with direct execution. Conceptually, however, it is a straightforward task to modify the emulator program to trace the execution of any application program it executes.
4. **Microcode modification.** In the days when processors executed microcode to execute their instruction sets through interpretation, it was possible to modify the microcode to generate a trace of each instruction executed. One important advantage of this approach was that it traced every instruction executed on the processor, including operating-system code. This feature was especially useful for tracing entire systems, including the interaction between the application programs and the operating system. The lack of microcode on current processors severely limits the applicability of this approach today.
5. **Compiler modification.** Another approach for generating traces is to modify the executable code produced by the compiler. Similar to what must be done for generating basic-block counts, extra instructions are added at the start of each basic block to record when the block is entered and which basic block is being executed then. Details about the contents of the basic blocks can be obtained from the compiler and correlated to the dynamic basic-block trace to produce a complete trace of all of the instructions executed by the application program. It is possible to add this type of tracing facility as a compilation option, or to write a post-compilation software tool that modifies the executable program generated by the compiler.

These trace-generation techniques are by no means the only ways in which traces can be produced. Rather, they are intended to give you a flavor of the types of approaches that have been used successfully in other trace-generation systems. Indeed, new techniques are limited only by the imagination and creativity of the performance analyst.

#### 6.4.2 Trace compression

One obvious concern when generating a trace is the execution-time slowdown and other program perturbations caused by the execution of the additional tracing instructions. Another concern is the volume of data that can be produced in a very short time. For example, say we wish to trace every instruction executed by a processor that executes at an average rate of  $10^8$  instructions per second. If

each item in the trace requires 16 bits to encode the necessary information, our tracing will produce more than 190 Mbytes of data per uninstrumented second of execution time, or more than 11 Gbytes per minute! In addition to obtaining the disks necessary to store this amount of data, the input/output operations required to move this large volume of data from the traced program to the disks create additional perturbations. Thus, it is desirable to reduce the amount of information that must be stored.

#### **6.4.2.1 Online trace consumption**

One approach for dealing with these large data volumes is to consume the trace *online*. That is, instead of storing the trace for later use, the program that will be driven by the trace is run simultaneously with the application program being traced. In this way, the trace is consumed as it is generated so that it never needs to be stored on disk at all.

A potential problem with online trace consumption in a multitasked (i.e. time-shared) system is the potential interdeterminate behavior of the program being traced. Since system events occur asynchronously with respect to the traced program, there is no assurance that the next time the program is traced the exact same sequence of events will occur in the same relative time order. This is a particular concern for programs that must respond to real-time events, such as system interrupts and user inputs.

This potential lack of repeatability in generating the trace is a concern when performing one-to-one comparison experiments. In this situation, the trace-consumption program is driven once with the trace and its output values are recorded. It is then modified in some way and then driven again with the same trace. If the identical input trace is used both times, it is reasonable to conclude that any change in performance observed is due to the change made to the trace-consumption program. However, if it cannot be guaranteed that the trace is identical from one run to the next, it is not possible to determine whether any change in performance observed is due to the change made, or whether it is due to a difference in the input trace itself.

#### **6.4.2.2 Compression of data**

A trace written to intermediate storage, such as a disk, can be viewed just like any other type of data file. Consequently, it is quite reasonable to apply a data-compression algorithm to the trace data as it is written to the disk. For example, any one of the large number of compression programs based on the popular Lempel–Ziv algorithm is often able to reduce the size of a trace file by 20–70%. Of course, the tradeoff for this data compression is the additional time required to execute the compression routine when the trace is generated and the time required to uncompress the trace when it is consumed.

#### 6.4.2.3 Abstract execution

An interesting variation of the basic trace-compression idea takes advantage of the semantic information within a program to reduce the amount of information that must be stored for a trace. This approach, called *abstract execution*, separates the tracing process into two steps. The first step performs a compiler-style analysis of the program to be traced. This analysis identifies a small subset of the entire trace that is sufficient to later reproduce the full trace. Only this smaller subset is actually stored. Later, the trace-consumption program must execute some special trace-regeneration routines to convert this partial trace information into the full trace. These regeneration routines are automatically generated by the tracing tool when it performs the initial analysis of the program.

The data about the full trace that are actually stored when using the abstract-execution model consist of information describing only those transitions that may change during run-time. For example, consider the code fragment extracted from a program to be traced shown in Figure 6.9. The compiler-style analysis that would be performed on this code fragment would produce the control flow graph shown in Figure 6.10. From this control flow graph, the trace-generation tool can determine that statement 1 always precedes both statements 2 and 3. Furthermore, statement 4 always follows both statements 2 and 3. When this program is executed, the trace through this sequence of statements will be either 1–2–4, or 1–3–4. Thus, the only information that needs to be recorded during run-time is which of statements 2 and 3 actually occurred. The trace-regeneration routine is then able to later reconstruct the full trace using the previously recorded control flow graph.

Measurements of the effectiveness of this tracing technique have shown that it slows down the execution of the program being traced by a factor of typically 2–10. This slowdown factor is comparable to, or slightly better than, those of most other tracing techniques. More important, however, may be that, by recording information only about the changes that actually occur during run-time, this technique is able to reduce the size of the stored traces by a factor of ten to several hundred.

```
1. if (i > 5)
2.     then a = a + i;
3.     else b = b + 1;
4. i = i + 1;
```

Figure 6.9 A code fragment to be processed using the abstract execution tracing technique..

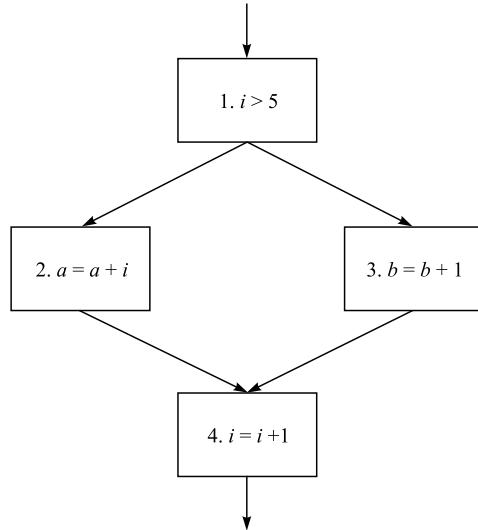


Figure 6.10 The control flow graph corresponding to the program fragment shown in Figure 6.9.

#### 6.4.2.4 Trace sampling

*Trace sampling* is another approach that has been suggested for reducing the amount of information that must be collected and stored when tracing a program. The basic idea is to save only relatively small sequences of events from locations scattered throughout the trace. The expectation is that these small samples will be statistically representative of the entire program's trace when they are used. For instance, using these samples to drive a simulation should produce overall results that are similar to what would be produced if the simulation were to be driven with the entire trace.

Consider the sequence of events from a trace shown in Figure 6.11. Each sample from this trace consists of  $k$  consecutive events. The number of events between the starts of consecutive samples is the *sampling interval*, denoted by  $P$ . Since only the samples from the trace are actually recorded, the total amount of storage required for the trace can be reduced substantially compared with storing the entire raw trace.

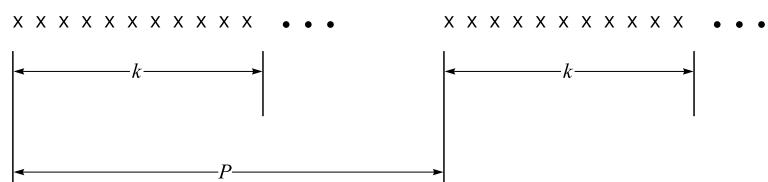


Figure 6.11 In trace sampling,  $k$  consecutive events comprise one sample of the trace. A new sample is taken every  $P$  events ( $P$  is called the sampling interval).

Unfortunately, there is no solid theoretical basis to help the experimenter determine how many events should be stored for each sample ( $k$ ), or how large the sampling interval ( $P$ ) should be. The best choices for  $k$  and  $P$  typically must be determined empirically (i.e. through experimentation). Furthermore, the choice of these parameters seems to be dependent on how the traces will be used. If the traces are used to drive a simulation of a cache to estimate cache-miss ratios, for instance, it has been suggested (see Laha *et al.* (1988)) that, in a trace of tens of millions of memory references, it is adequate to have several thousand events per sample. The corresponding sampling interval then should be chosen to provide enough samples such that 5–10% of the entire trace is recorded. These results, however, appear to be somewhat dependent on the size of the cache being simulated. The bottom line is that, while trace sampling appears to be a reasonable technique for reducing the size of the trace that must be stored, a solid theoretical basis still needs to be developed before it can be considered ‘standard practice.’

---

## 6.5 Indirect and *ad hoc* measurements

---

Sometimes the performance metric we need is difficult, if not impossible, to measure directly. In this case, we have to rely on our ingenuity to develop an *ad hoc* technique to somehow derive the information indirectly. For instance, perhaps we are not able to directly measure the desired quantity, but we may be able to measure another related value directly. We may then be able to deduce the desired value from these other measured values.

For example, suppose that we wish to determine how much load a particular application program puts on a system when it is executed. We then may want to make changes to the program to see how they affect the system load. The first question we need to confront in this experiment is that of establishing a definition for the ‘system load.’

There are many possible definitions of the system load, such as the number of jobs on the run queue waiting to be executed, to name but one. In our case, however, we are interested in how much of the processor’s available time is spent executing our application program. Thus, we decide to define the average system load to be the fraction of time that the processor is busy executing users’ application programs.

If we had access to the source code of the operating system, we could directly measure this time by modifying the process scheduler. However, it is unlikely that we will have access to this code. An alternative approach is to directly measure how much time the processor spends executing an ‘idle’ process that we create. We then use this direct measurement of idle time to deduce how much

time the processor must have been busy executing real application programs during the given measurement interval.

Specifically, consider an ‘idle’ program that simply counts up from zero for a fixed period of time. If this program is the only application running on a single processor of a time-shared system, the final count value at the end of the measurement interval is the value that indirectly corresponds to an unloaded processor. If two applications are executed simultaneously and evenly share the processor, however, the processor will run our idle measurement program half as often as when it was the only application running. Consequently, if we allow both programs to run for the same time interval as when we ran the idle program by itself, its total count value at the end of the interval should be half of the value observed when only a single copy was executed.

Similarly, if three applications are executed simultaneously and equally share the processor for the same measurement interval, the final count value in our idle program should be one-third of the value observed when it was executed by itself. This line of thought can be further extended to  $n$  application programs simultaneously sharing the processor. After calibrating the counter process by running it by itself on an otherwise unloaded system, it can be used to indirectly measure the system load.

**Example.** In a time-shared system, the operating system will share a single processor evenly among all of the jobs executing in the system. Each available job is allowed to run for the *time slice*  $T_s$ . After this interval, the currently executing job is temporarily put to sleep, and the next ready job is switched in to run. Indirect load monitoring takes advantage of this behavior to estimate the system load. Initially, the load-monitor program is calibrated by allowing it to run by itself for a time  $T$ , as shown in Figure 6.12(a). At the end of this time, its counter value,  $n$ , is recorded. If the load monitor and another application are run simultaneously so that in total two jobs are sharing the processor, as shown in Figure 6.12(b), each job would be expected to be executing for half of the total time available. Thus, if the load monitor is again allowed to run for time  $T$ , we would expect its final count value to be  $n/2$ . Similarly, running the load monitor with two other applications for time  $T$  would result in a final count value of  $n/3$ , as shown in Figure 6.12(c). Consequently, knowing the value of the count after running the load monitor for time  $T$  allows us to deduce what the average load during the measurement interval must have been ◇

---

## 6.6 Perturbations due to measuring

---

One of the curious (and certainly most annoying!) aspects of developing tools to measure computer-systems performance is that instrumenting a system or pro-

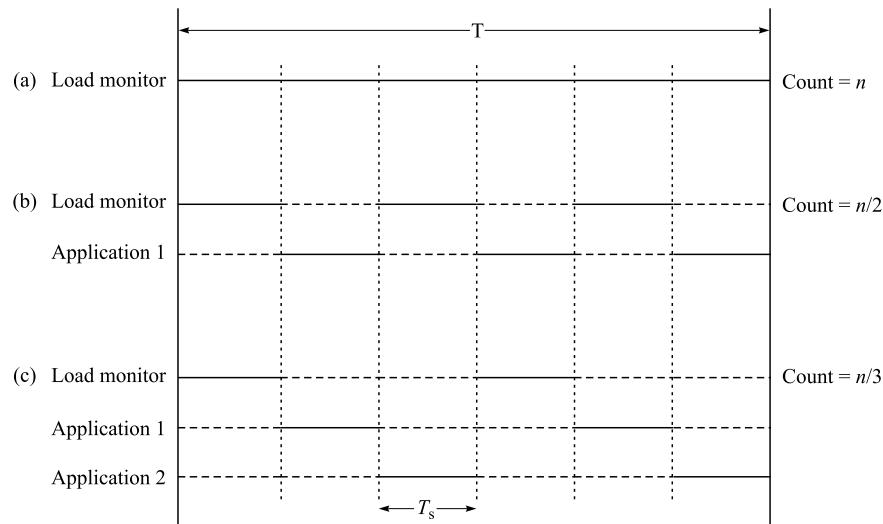


Figure 6.12 An example of using an indirect measurement technique to estimate the average system load in a time-shared system. The solid lines indicate when each application is running.

gram changes what we are trying to measure. Obtaining more information, or obtaining higher resolution measurements, for instance, requires more instrumentation points in a program. However, more instrumentation causes there to be more perturbations in the program than there are in its uninstrumented execution behavior. These additional perturbations due to the additional instrumentation then make the data we collect less reliable. As a result, we are almost always forced to use insufficient data to infer the behavior of the system in which we are interested.

To further confound the situation, performance perturbations due to instrumentation are nonlinear and nonadditive. They are nonlinear in the sense that doubling the amount of instrumentation in a program will not necessarily double its impact on performance, for instance. Similarly, instrumentation perturbation is nonadditive in the sense that adding more instrumentation can cancel out the perturbation effects of other instrumentation. Or, in some situations, additional instrumentation can multiplicatively increase the perturbations.

For example, adding code to an application program to generate an instruction trace can significantly change the spatial and temporal patterns of its memory accesses. The trace-generation code will cause a large number of extra store instructions to be executed, for instance, which can cause the cache to be effectively flushed at each trace point. These frequent cache flushes will then increase the number of cache misses, which will substantially impact the overall performance. If additional instrumentation is added, however, it may be possible that

---

the additional memory locations necessary for the instrumentation could change the pattern of conflict misses in the cache in such a way as to actually improve the cache performance perceived by the application. The bottom line is that the effects of adding instrumentation to a system being tested are entirely unpredictable.

Besides these direct changes to a program's performance, instrumenting a program can cause more subtle indirect perturbations. For example, an instrumented program will take longer to execute than will the uninstrumented program. This increase in execution time will then cause it to experience more context switches than it would have experienced if it had not been instrumented. These additional context switches can substantially alter the program's paging behavior, for instance, making the instrumented program behave substantially differently than the uninstrumented program.

---

**6.7****Summary**

Event-driven measurement tools record information about the system being tested whenever some predefined event occurs, such as a page fault or a network operation, for instance. The information recorded may be a simple count of the number of times the event occurred, or it may be a portion of the system's state at the time the event occurred. A time-ordered list of this recorded state information is called a trace. While event-driven tools record all occurrences of the defined events, sampling tools query some aspect of the system's state at fixed time intervals. Since this sampling approach will not record every event, it provides a statistical view of the system. Indirect measurement tools are used to deduce some aspect of a system's performance that it is difficult or impossible to measure directly.

Some perturbation of a system's behavior due to instrumentation is unavoidable. Furthermore, and more difficult to compensate for, perhaps, is the unpredictable relationship between the instrumentation and its impact on performance. Through experience and creative use of measurement techniques, the performance analyst can try to minimize the impact of these perturbations, or can sometimes compensate for their effects.

It is important to bear in mind, though, that measuring a system alters it. While you would like to measure a completely uninstrumented program, what you actually end up measuring is the instrumented system. Consequently, you must always remain alert to how these perturbations may bias your measurements and, ultimately, the conclusions you are able to draw from your experiments.



## Chapter 8

# Topics in Distributed Coordination and Distributed Transactions

This chapter contains parts of the book chapter:

G. Coulouris, J. Dollimore, T. Kindberg. Distributed systems, concepts and design. Third Edition. Chapters 11 (except 11.2 and 11.3) and 13, pp. 419–423, 436–464, and 515–552 (72 of 772). Addison-Wesley ,2001. ISBN: 0201-61918-0

In addition, the chapter also contains the paper:

F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4) pp. 299–319 (21 of 409), 1990. Doi: 10.1145/98163.98167

Achieving both **high availability** and **atomicity** simultaneously is a hard problem. As we have seen previously, two strategies employed for availability are redundancy via replication and distribution of system functionality. When these techniques are employed, atomicity of our higher-level service is typically compromised. An important question is whether atomicity can be reestablished even when replication or distribution are employed. The text explores techniques from distributed systems to provide us with an only partially positive answer, which depends fundamentally on the assumptions made about the underlying failure model. First, the text explores the primitive of multicast communication, which can be used as a building block in a replication protocol. Second, we turn to distributed transactions, which can be used to atomically interact with a system with distributed functionality. *The ultimate goal of this portion of the material is to provides us with an overview of the results in distributed systems related to synchronous replication and distributed transactions, as well as basic protocols to achieve both, in particular totally-ordered multicast and two-phase commit.*

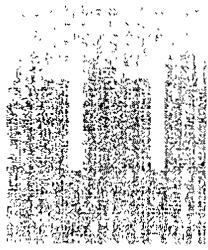
The learning goals for this portion of the material are listed below.

- Explain the difficulties of guaranteeing atomicity in a replicated distributed system.

- Explain the notion of state-machine replication and the ISIS approach to totally ordered multicast among replicas.
- Describe the implications of the FLP impossibility result and possible workarounds.
- Explain mechanisms necessary for distributed transactions, such as distributed locking and distributed recovery.
- Explain the operation of the two-phase commit protocol (2PC).
- Predict outcomes of 2PC under failure scenarios.

*Schneider's tutorial paper on state-machine replication is given to deepen understanding of this approach to replication; however, it is to be considered as an additional reading and not fundamental to the attainment of the learning goals above.*

*By contrast, particular attention should be paid above to the material on totally-ordered multicast (Section 11.4.3) and the overview of the consensus problem and its related theoretical results (Sections 11.5.1, 11.5.2, and 11.5.4), as well as to the material on distributed transactions, especially the two-phase commit protocol (Coulouris et al.'s Chapter 13, especially Section 13.3).*



# COORDINATION AND AGREEMENT

- 11.1 Introduction
- 11.2 Distributed mutual exclusion
- 11.3 Elections
- 11.4 Multicast communication
- 11.5 Consensus and related problems
- 11.6 Summary

In this chapter, we introduce some topics and algorithms related to the issue of how processes coordinate their actions and agree on shared values in distributed systems, despite failures. The chapter begins with algorithms to achieve mutual exclusion among a collection of processes, so as to coordinate their accesses to shared resources. It goes on to examine how an election can be implemented in a distributed system. That is, it describes how a group of processes can agree on a new coordinator of their activities after the previous coordinator has failed.

The second half examines the related problems of multicast communication, consensus, byzantine agreement and interactive consistency. In multicast, the issue is how to agree on such matters as the order in which messages are to be delivered. Consensus and the other problems generalize from this: how can any collection of processes agree on some value, no matter what the domain of the values in question? We encounter a fundamental result in the theory of distributed systems: that under certain conditions – including surprisingly benign failure conditions – it is impossible to guarantee that processes will reach consensus.

## 11.1 Introduction

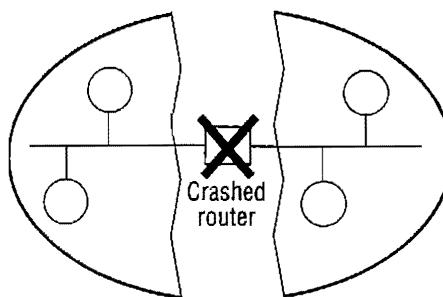
This chapter introduces a collection of algorithms whose goals vary but that share an aim that is fundamental in distributed systems: for a set of processes to coordinate their actions or to agree on one or more values. For example, in the case of a complex piece of machinery such as a spaceship, it is essential that the computers controlling it agree on such conditions as whether the spaceship's mission is proceeding or has been aborted. Furthermore, the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators). The computers must be able to do so even where there is no fixed master-slave relationship between the components (which would make coordination particularly simple). The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.

An important distinction for us, as in Chapter 10, will be whether the distributed system under study is asynchronous or synchronous. In an asynchronous system we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, on the time to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.

Another important aim of the chapter while discussing algorithms is to consider failures, and how to deal with them when designing algorithms. Section 2.3.2 introduced a failure model, which we shall use in this chapter. Coping with failures is a subtle business, so we begin by considering some algorithms that tolerate no failures and progress through benign failures until we consider how to tolerate arbitrary failures. We encounter a fundamental result in the theory of distributed systems. Even under surprisingly benign failure conditions, it is impossible to guarantee in an asynchronous system that a collection of processes can agree on a shared value – for example, for all of a spaceship's controlling processes to agree 'mission proceed' or 'mission abort'.

Section 11.2 examines the problem of distributed mutual exclusion. This is the extension to distributed systems of the familiar problem of avoiding race conditions in kernels and multi-threaded applications. Since much of what occurs in distributed systems is resource sharing, this is an important problem to solve. Next, Section 11.3 introduces a related but more general issue of how to 'elect' one of a collection of processes to perform a special role. For example, in Chapter 10 we saw how processes synchronized their clocks to a designated time server. If this server fails and several surviving servers can fulfil that role, then for the sake of consistency it is necessary to choose just one server to take over.

Multicast communication is the subject of Section 11.4. As Section 4.5.1 explained, multicast is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data. Section 11.4 examines multicast reliability and ordering semantics, and gives algorithms to achieve the variations. Multicast delivery is essentially a problem of agreement between processes: the recipients agree on which messages they will receive, and in which order they will receive them. Section 11.5 discusses the problem of agreement more generally, primarily in the forms known as consensus and byzantine agreement.

**Figure 11.1** A network partition

The treatment followed in this chapter involves stating the assumptions and the goals to be met, and giving an informal account of why the algorithms presented are correct. There is insufficient space to provide a more rigorous approach. For that, we refer the reader to a text that gives a thorough account of distributed algorithms, such as Attiya and Welch [1998] and Lynch [1996].

Before presenting the problems and algorithms, we discuss failure assumptions and the practical matter of detecting failures in distributed systems.

### 11.1.1 Failure assumptions and failure detectors

For the sake of simplicity, this chapter assumes that each pair of processes is connected by reliable channels. That is, although the underlying network components may suffer failures, the processes use a reliable communication protocol that masks these failures – for example, by retransmitting missing or corrupted messages. Also for the sake of simplicity, we assume that no process failure implies a threat to the other processes’ ability to communicate. This means that none of the processes depends upon another to forward messages.

Note that a reliable channel *eventually* delivers a message to the recipient’s input buffer. In a synchronous system, we suppose that there is hardware redundancy where necessary, so that a reliable channel not only eventually delivers each message despite underlying failures but it does so within a specified time bound.

In any particular interval of time, communication between some processes may succeed while communication between others is delayed. For example, the failure of a router between two networks may mean that a collection of four processes is split into two pairs, such that intra-pair communication is possible over their respective networks; but inter-pair communication is not possible while the router has failed. This is known as a *network partition* (Figure 11.1). Over a point-to-point network such as the Internet, complex topologies and independent routing choices mean that connectivity may be *asymmetric*: communication is possible from process  $p$  to process  $q$ , but not *vice versa*. Connectivity may also be *intransitive*: communication is possible from  $p$  to  $q$  and from  $q$  to  $r$ ; but  $p$  cannot communicate directly with  $r$ . Thus our reliability assumption entails that eventually any failed link or router will be repaired or circumvented. Nevertheless, the processes may not all be able to communicate at the same time.

The chapter assumes, unless we state otherwise, that processes only fail by crashing – an assumption that is good enough for many systems. In Section 11.5, we shall consider how to treat the cases where processes have arbitrary (byzantine) failures. Whatever the type of failure, a *correct* process is one that exhibits no failures at any point in the execution under consideration. Note that correctness applies to the whole execution, not just to a part of it. So a process that suffers a crash failure is ‘non-failed’ before that point, not ‘correct’ before that point.

One of the problems in the design of algorithms that can overcome process crashes is that of deciding when a process has crashed. A *failure detector* [Chandra and Toueg 1996, Stelling *et al.* 1998] is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a *local failure detector*. We shall outline how to implement failure detectors shortly, but first we shall concentrate on some of the properties of failure detectors.

A failure ‘detector’ is not necessarily accurate. Most fall into the category of *unreliable failure detectors*. An unreliable failure detector may produce one of two values when given the identity of a process: *Unsuspected* or *Suspected*. Both of these results are hints, which may or may not accurately reflect whether the process has actually failed. A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed; for example, a message was recently received from it. But of course the process can have failed since then. A result of *Suspected* signifies that the failure detector has some indication that the process may have failed. For example, it may be that no message from the process has been received for more than a nominal maximum length of silence (even in an asynchronous system, practical upper bounds can be used as hints). The suspicion may be misplaced: for example, the process could be functioning correctly, but on the other side of a network partition; or it could be running more slowly than expected.

A *reliable failure detector* is one that is always accurate in detecting a process’s failure. It answers processes’ queries with either a response of *Unsuspected* – which, as before, can only be a hint – or *Failed*. A result of *Failed* means that the detector has determined that the process has crashed. Recall that a process that has crashed stays that way, since by definition a process never takes another step once it has crashed.

It is important to realize that, although we speak of one failure detector acting for a collection of processes, the response that the failure detector gives to a process is only as good as the information available at that process. A failure detector may sometimes give different responses to different processes, since communication conditions vary from process to process.

We can implement an unreliable failure detector using the following algorithm. Each process  $p$  sends a ‘ $p$  is here’ message to every other process, and it does this every  $T$  seconds. The failure detector uses an estimate of the maximum message transmission time of  $D$  seconds. If the local failure detector at process  $q$  does not receive a ‘ $p$  is here’ message within  $T + D$  seconds of the last one, then it reports to  $q$  that  $p$  is *Suspected*. However, if it subsequently receives a ‘ $p$  is here’ message, then it reports to  $q$  that  $p$  is *OK*.

In a real distributed system, there are practical limits on message transmission times. Even email systems give up after a few days, since it is likely that communication

links and routers will have been repaired in that time. If we choose small values for  $T$  and  $D$  (so that they total 0.1 second, say), then the failure detector is likely to suspect non-crashed processes many times, and much bandwidth will be taken up with ‘ $p$  is here’ messages. If we choose a large total timeout value (a week, say) then crashed processes will often be reported as *Unsuspected*.

A practical solution to this problem is to use timeout values that reflect the observed network delay conditions. If a local failure detector receives a ‘ $p$  is here’ in 20 seconds instead of the expected maximum of 10 seconds, then it could reset its timeout value for  $p$  accordingly. The failure detector remains unreliable, and its answers to queries are still only hints, but the probability of its accuracy increases.

In a synchronous system, our failure detector can be made into a reliable one. We can choose  $D$  so that it is not an estimate but an absolute bound on message transmission times; the absence of a ‘ $p$  is here’ message within  $T + D$  seconds entitles the local failure detector to conclude that  $p$  has crashed.

The reader may wonder whether failure detectors are of any practical use. Unreliable failure detectors may suspect a process that has not failed (they may be *inaccurate*); and they may not suspect a process that has in fact failed (they may be *incomplete*). Reliable failure detectors, on the other hand, require that the system is synchronous (and few practical systems are).

We have introduced failure detectors because they help us to think about the nature of failures in a distributed system. And any practical system that is designed to cope with failures must detect them – however imperfectly. But it turns out that even unreliable failure detectors with certain well-defined properties can help us to provide practical solutions to the problem of coordinating processes in the presence of failures. We return to this point in Section 11.5.

## 11.2 Distributed mutual exclusion

Distributed processes often need to coordinate their activities. If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is the *critical section* problem, familiar in the domain of operating systems. In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general. We require a solution to *distributed mutual exclusion*: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion. Chapter 12 describes how some servers synchronize client accesses to resources. But in some practical cases a separate mechanism for mutual exclusion is required.

Consider users who update a text file. A simple means of ensuring that their updates are consistent is to allow them to access it only one at a time, by requiring the editor to lock the file before updates can be made. NFS file servers, described in Chapter 8, are designed to be stateless and therefore do not support file locking. For this reason, UNIX systems provide a separate file-locking service, implemented by the daemon *lockd*, to handle locking requests from clients.

Taking the example just given, suppose that  $p_3$  either had not failed but was running unusually slowly (that is, the assumption that the system is synchronous is incorrect) or that  $p_3$  had failed but is then replaced. Just as  $p_2$  sends its *coordinator* message,  $p_3$  (or its replacement) does the same.  $p_2$  receives  $p_3$ 's *coordinator* message after it sent its own and so sets  $elected_2 = p_3$ . Due to variable message transmission delays,  $p_1$  receives  $p_2$ 's *coordinator* message after  $p_3$ 's and so eventually sets  $elected_1 = p_2$ . Condition E1 has been broken.

With regard to the performance of the algorithm, in the best case the process with the second highest identifier notices the coordinator's failure. Then it can immediately elect itself and send  $N - 2$  coordinator messages. The turnaround time is one message. The bully algorithm requires  $O(N^2)$  messages in the worst case – that is, when the process with the least identifier first detects the coordinator's failure. For then  $N - 1$  processes altogether begin elections, each sending messages to processes with higher identifiers.

## 11.4 Multicast communication

Section 4.5.1 described IP multicast, which is an implementation of group communication. Group, or multicast, communication requires coordination and agreement. The aim is for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees. The guarantees include agreement on the set of messages that every process in the group should receive and on the delivery ordering across the group members.

Group communication systems are extremely sophisticated. Even IP multicast, which provides minimal delivery guarantees, requires a major engineering effort. Time and bandwidth efficiency are important concerns, and are challenging even for static groups of processes. The problems are multiplied when processes can join and leave groups at arbitrary times.

Here we study multicast communication to groups of processes whose membership is known. Chapter 14 will expand our study to fully fledged group communication, including the management of dynamically varying groups.

Multicast communication has been the subject of many projects, including the V-system [Cheriton and Zwaenepoel 1985], Chorus [Rozier *et al.* 1988], Amoeba [Kaashoek *et al.* 1989, Kaashoek and Tanenbaum 1991], Trans/Total [Melliar-Smith *et al.* 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse *et al.* 1996], Totem [Moser *et al.* 1996] and Transis [Dolev and Malki 1996] – and we shall cite other notable work in the course of this section.

The essential feature of multicast communication is that a process issues only one *multicast* operation to send a message to each of a group of processes (in Java this operation is `aSocket.send(aMessage)`) instead of issuing multiple *send* operations to individual processes. Communication to *all* processes in the system, as opposed to a sub-group of them, is known as *broadcast*.

The use of a single *multicast* operation instead of multiple *send* operations amounts to much more than a convenience for the programmer. It enables the

implementation to be efficient and allows it to provide stronger delivery guarantees than would otherwise be possible.

*Efficiency:* The information that the same message is to be delivered to all processes in a group allows the implementation to be efficient in its utilization of bandwidth. It can take steps to send the message no more than once over any communication link, by sending the message over a distribution tree; and it can use network hardware support for multicast where this is available. The implementation can also minimize the total time taken to deliver the message to all destinations, instead of transmitting it separately and serially.

To see these advantages, compare the bandwidth utilization and the total transmission time taken when sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto, (a) by two separate UDP sends, and (b) by a single IP-multicast operation. In the former case, two copies of the messages are sent independently, and the second is delayed by the first. In the latter case, a set of multicast-aware routers forward a single copy of the message from London to a router on the destination LAN. The final router then uses hardware multicast (provided by the Ethernet) to deliver the message to the destinations, instead of sending it twice.

*Delivery guarantees:* If a process issues multiple independent *send* operations to individual processes, then there is no way for the implementation to provide delivery guarantees that affect the group of processes as a whole. If the sender fails half-way through sending, then some members of the group may receive the message while others do not. And the relative ordering of two messages delivered to any two group members is undefined. In the particular case of IP multicast, no ordering or reliability guarantees are in fact offered. But stronger multicast guarantees can be made, and we shall shortly define some.

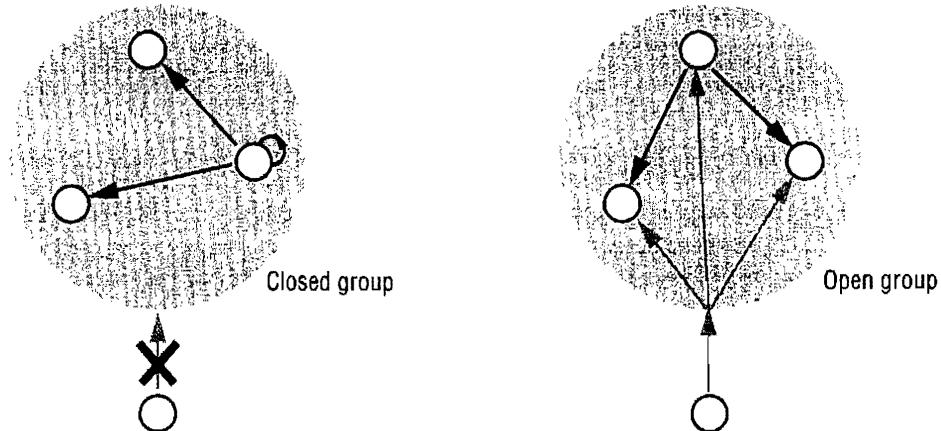
**System model** ◊ The system contains a collection of processes, which can communicate reliably over one-to-one channels. As before, processes may fail only by crashing.

The processes are members of groups, which are the destinations of messages sent with the *multicast* operation. It is generally useful to allow processes to be members of several groups simultaneously – for example, to enable processes to receive information from several sources by joining several groups. But to simplify our discussion of ordering properties, we shall sometimes restrict processes to being members of at most one group at a time.

The operation *multicast*( $g, m$ ) sends the message  $m$  to all members of the group  $g$  of processes. Correspondingly, there is an operation *deliver*( $m$ ) that delivers a message sent by multicast to the calling process. We use the term *deliver* rather than *receive* to make clear that a multicast message is not always handed to the application layer inside the process as soon as it is received at the process's node. This is explained when we discuss multicast delivery semantics shortly.

Every message  $m$  carries the unique identifier of the process *sender*( $m$ ) that sent it, and the unique destination group identifier *group*( $m$ ). We assume that processes do not lie about the origin or destinations of messages.

A group is said to be *closed* if only members of the group may multicast to it (Figure 11.9). A process in a closed group delivers to itself any message that it

**Figure 11.9** Open and closed groups

multicasts to the group. A group is *open* if processes outside the group may send to it. (The categories ‘open’ and ‘closed’ also apply with analogous meanings to mailing lists.) Closed groups of processes are useful, for example, for cooperating servers to send messages to one another that only they should receive. Open groups are useful, for example, for delivering events to groups of interested processes.

Some algorithms assume that groups are closed. The same effect as openness can be achieved with a closed group by picking a member of the group and sending it a message (one-to-one) for it to multicast to its group. Rodrigues *et al.* [1998] discuss multicast to open groups.

#### 11.4.1 Basic multicast

It is useful to have at our disposal a basic multicast primitive that guarantees, unlike IP multicast, that a correct process will eventually deliver the message, as long as the multicaster does not crash. We call the primitive *B-multicast* and its corresponding basic delivery primitive is *B-deliver*. We allow processes to belong to several groups, and each message is destined for some particular group.

A straightforward way to implement *B-multicast* is to use a reliable one-to-one *send* operation, as follows:

To *B-multicast*( $g, m$ ): for each process  $p \in g$ , *send*( $p, m$ );

On *receive*( $m$ ) at  $p$ : *B-deliver*( $m$ ) at  $p$ .

The implementation may use threads to perform the *send* operations concurrently, in an attempt to reduce the total time taken to deliver the message. Unfortunately, such an implementation is liable to suffer from a so-called *ack-implosion* if the number of processes is large. The acknowledgments sent as part of the reliable *send* operation are liable to arrive from many processes at about the same time. The multicasting process’s buffers will rapidly fill and it is liable to drop acknowledgments. It will therefore retransmit the message, leading to yet more acknowledgments and further waste of

network bandwidth. A more practical basic multicast service can be built using IP multicast, and we invite the reader to show this.

### 11.4.2 Reliable multicast

Section 2.3.2 defined reliable one-to-one communication channels between pairs of processes. The required safety property is called *integrity* – that any message delivered is identical to one that was sent, and that no message is delivered twice. The required liveness property is called *validity*: that any message is eventually delivered to the destination, if it is correct.

Following Hadzilacos and Toueg [1994] and Chandra and Toueg [1996], we now define a *reliable multicast*, with corresponding operations *R-multicast* and *R-deliver*. Properties analogous to integrity and validity are clearly highly desirable in reliable multicast delivery. But we add another: a requirement that *all* correct processes in the group must receive a message if *any* of them does. It is important to realize that this is not a property of the *B-multicast* algorithm that is based on a reliable one-to-one send operation. The sender may fail at any point while *B-multicast* proceeds, so some processes may deliver a message while others do not.

A reliable multicast is one that satisfies the following properties; we explain the properties after stating them.

*Integrity:* A correct process  $p$  delivers a message  $m$  at most once. Furthermore,  $p \in \text{group}(m)$  and  $m$  was supplied to a *multicast* operation by  $\text{sender}(m)$ . (As with one-to-one communication, messages can always be distinguished by a sequence number relative to their sender.)

*Validity:* If a correct process multicasts message  $m$  then it will eventually deliver  $m$ .

*Agreement:* If a correct process delivers message  $m$ , then all other correct processes in  $\text{group}(m)$  will eventually deliver  $m$ .

The integrity property is analogous to that for reliable one-to-one communication. The validity property guarantees liveness for the sender. This may seem an unusual property, because it is asymmetric (it mentions only one particular process). But notice that validity and agreement together amount to an overall liveness requirement: if one process (the sender) eventually delivers a message  $m$  then, since the correct processes agree on the set of messages they deliver, it follows that  $m$  will eventually be delivered to all the group's correct members.

The advantage of expressing the validity condition in terms of self-delivery is simplicity. What we require is that the message be delivered eventually by *some* correct member of the group.

The agreement condition is related to atomicity, the property of ‘all or nothing’, applied to delivery of messages to a group. If a process that multicasts a message crashes before it has delivered it, then it is possible that the message will not be delivered to any process in the group; but if it is delivered to *some* correct process, then all other correct processes will deliver it. Many papers in the literature use the term ‘atomic’ to include a total ordering condition; we define this shortly.

**Figure 11.10** Reliable multicast algorithm

---

*On initialization*

*Received* := {};

*For process p to R-multicast message m to group g*

*B-multicast(g, m); // p ∈ g is included as a destination*

*On B-deliver(m) at process q with g = group(m)*

*if (m ∉ Received)*

*then*

*Received* := *Received* ∪ {m};

*if (q ≠ p) then B-multicast(g, m); end if*

*R-deliver m;*

*end if*

---

**Implementing reliable multicast over B-multicast** ◊ Figure 11.10 gives a reliable multicast algorithm, with primitives *R-multicast* and *R-deliver*, which allows processes to belong to several closed groups simultaneously. To *R-multicast* a message, a process *B-multicasts* the message to the processes in the destination group (including itself). When the message is *B-delivered*, the recipient in turn *B-multicasts* the message to the group (if it is not the original sender), and then *R-delivers* the message. Since a message may arrive more than once, duplicates of the message are detected and not delivered.

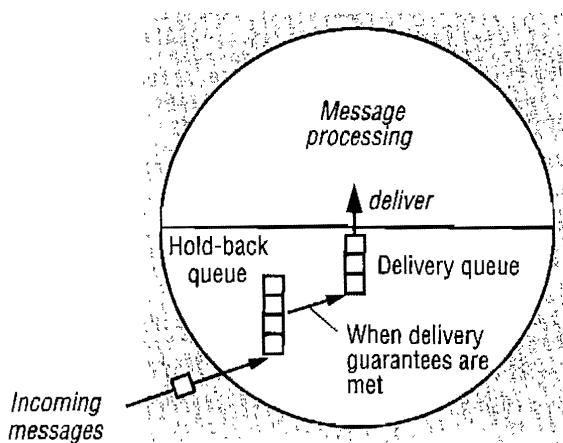
This algorithm clearly satisfies validity, since a correct process will eventually *B-deliver* the message to itself. By the integrity property of the underlying communication channels used in *B-multicast*, the algorithm also satisfies the integrity property.

Agreement follows from the fact that every correct process *B-multicasts* the message to the other processes after it has *B-delivered* it. If a correct process does not *R-deliver* the message, then this can only be because it never *B-delivered* it. That in turn can only be because no other correct process *B-delivered* it either; therefore, none will *R-deliver* it.

The reliable multicast algorithm that we have described is correct in an asynchronous system, since we made no timing assumptions. But the algorithm is inefficient for practical purposes. Each message is sent  $|g|$  times to each process.

**Reliable multicast over IP multicast** ◊ An alternative realization of *R-multicast* is to use a combination of IP multicast, piggy backed acknowledgments (that is, acknowledgements attached to other messages), and negative acknowledgments. This *R-multicast* protocol is based on the observation that IP multicast communication is often successful. In the protocol, processes do not send separate acknowledgment messages; instead, they piggy back acknowledgments on the messages that they send to the group. Processes send a separate response message only when they detect that they have missed a message. A response indicating the absence of an expected message is known as a *negative acknowledgement*.

The description assumes that groups are closed. Each process  $p$  maintains a sequence number  $S_g^p$  for each group  $g$  to which it belongs. The sequence number is initially zero. Each process also records  $R_g^q$ , the sequence number of the latest message it has delivered from process  $q$  that was sent to group  $g$ .

**Figure 11.11** The hold-back queue for arriving multicast messages

For  $p$  to  $R$ -multicast a message to group  $g$ , it piggy backs onto the message the value  $S_g^p$  and acknowledgments, of the form  $\langle q, R_g^q \rangle$ . An acknowledgement states, for some sender  $q$ , the sequence number of the latest message from  $q$  destined for  $g$  that  $p$  has delivered since it last multicast a message. The multicaster  $p$  then IP-multicasts the message with its piggy backed values to  $g$ , and increments  $S_g^p$  by one.

The piggy backed values in a multicast message enable the recipients to learn about messages that they have not received. A process  $R$ -delivers a message destined for  $g$  bearing the sequence number  $S$  from  $p$  if and only if  $S = R_g^p + 1$ , and it increments  $R_g^p$  by one immediately after delivery. If an arriving message has  $S \leq R_g^p$ , then  $r$  has delivered the message before and it discards it. If  $S > R_g^p + 1$ , or if  $R > R_g^q$  for an enclosed acknowledgement  $\langle q, R \rangle$ , then there are one or more messages that it has not yet received (and which are likely to have been dropped, in the first case). It keeps any message for which  $S > R_g^p + 1$  in a *hold-back queue* (Figure 11.11) – such queues are often used to meet message delivery guarantees. It requests missing messages by sending negative acknowledgements – to the original sender or to a process  $q$  from which it has received an acknowledgement  $\langle q, R_g^q \rangle$  with  $R_g^q$  no less than the required sequence number.

The hold-back queue is not strictly necessary for reliability but it simplifies the protocol by enabling us to use sequence numbers to represent sets of delivered messages. It also provides us with a guarantee of delivery order (see Section 11.4.3).

The integrity property follows from the detection of duplicates and the underlying properties of IP multicast (which uses checksums to expunge corrupted messages). The validity property holds because IP multicast has that property. For agreement we require, first, that a process can always detect missing messages. That in turn means that it will always receive a further message that enables it to detect the omission. As this simplified protocol stands, we guarantee detection of missing messages only in the case where correct processes multicast messages indefinitely. Second, the agreement property requires that there is always an available copy of any message needed by a process that did not receive it. We therefore assume that processes retain copies of the messages they have delivered – indefinitely, in this simplified protocol.

Neither of the assumptions we made to ensure agreement is practical (see Exercise 11.14). However, agreement is practically addressed in the protocols from which ours is derived: the Psync protocol [Peterson *et al.* 1989], Trans protocol [Melliar-Smith *et al.* 1990] and scalable reliable multicast protocol [Floyd *et al.* 1997]. Psync and Trans also provide further delivery ordering guarantees.

**Uniform properties** ◊ The definition of agreement given above refers only to the behaviour of *correct* processes – processes that never fail. Consider what would happen in the algorithm of Figure 11.10 if a process is not correct and crashed after it had *R-delivered* a message. Since any process that *R-delivers* the message must first *B-multicast* it, it follows that all correct processes will still eventually deliver the message.

Any property that holds whether or not processes are correct is called a *uniform* property. We define uniform agreement as follows:

*Uniform agreement:* If a process, whether it is correct or fails, delivers message  $m$ , then all correct processes in  $\text{group}(m)$  will eventually deliver  $m$ .

Uniform agreement allows a process to crash after it has delivered a message, while still ensuring that all correct processes will deliver the message. We have argued that the algorithm of Figure 11.10 satisfies this property, which is stronger than the non-uniform agreement property defined above.

Uniform agreement is useful in applications where a process may take an action that produces an observable inconsistency before it crashes. For example, consider that the processes are servers that manage copies of a bank account, and that updates to the account are sent using reliable multicast to the group of servers. If the multicast does not satisfy uniform agreement, then a client that accesses a server just before it crashes may observe an update that no other server will process.

It is interesting to note that if we reverse the lines ‘*R-deliver m*’ and ‘*if* ( $q \neq p$ ) *then B-multicast*( $g, m$ ); *end if*’ in Figure 11.10, then the resultant algorithm does not satisfy uniform agreement.

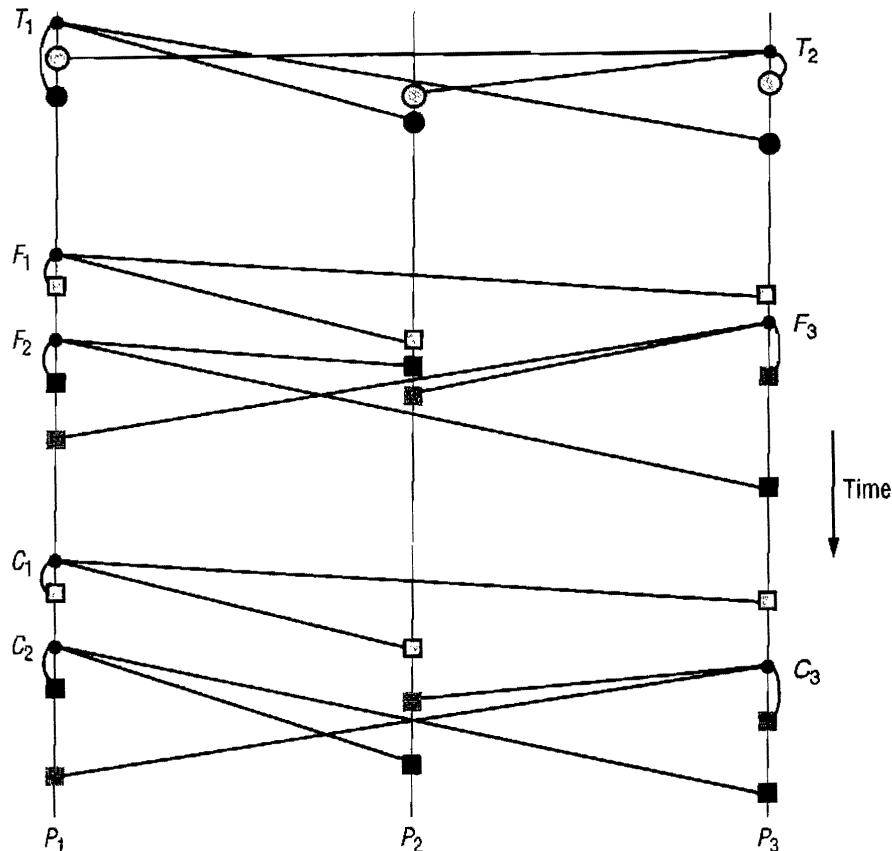
Just as there is a uniform version of agreement, there are also uniform versions of any multicast property, including validity and integrity and the ordering properties that we are about to define.

### 11.4.3 Ordered multicast

The basic multicast algorithm of Section 11.4.1 delivers messages to processes in an arbitrary order, due to arbitrary delays in the underlying one-to-one send operations. This lack of an ordering guarantee is not satisfactory for many applications. For example, in a nuclear power plant it may be important that events signifying threats to safety conditions and events signifying actions by control units are observed in the same order by all processes in the system.

The common ordering requirements are total ordering, causal ordering, FIFO ordering and the hybrids total-causal and total-FIFO. To simplify our discussion, we define these orderings under the assumption that any process belongs to at most one group. We shall later discuss the implications of allowing groups to overlap.

*FIFO ordering:* If a correct process issues *multicast*( $g, m$ ) and then *multicast*( $g, m'$ ), then every correct process that delivers  $m'$  will deliver  $m$  before  $m'$ .

**Figure 11.12** Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages  $T_1$  and  $T_2$ , the FIFO-related messages  $F_1$  and  $F_2$  and the causally related messages  $C_1$  and  $C_3$  – and the otherwise arbitrary delivery ordering of messages

*Causal ordering:* If  $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$ , where  $\rightarrow$  is the happened-before relation induced only by messages sent between the members of  $g$ , then any correct process that delivers  $m'$  will deliver  $m$  before  $m'$ .

*Total ordering:* If a correct process delivers message  $m$  before it delivers  $m'$ , then any other correct process that delivers  $m'$  will deliver  $m$  before  $m'$ .

Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by happened-before. Note that FIFO ordering and causal ordering are only partial orderings: not all messages are sent by the same process, in general; similarly, some multicasts are concurrent (not ordered by happened-before).

Figure 11.12 illustrates the orderings for the case of three processes. Close inspection of the figure shows that the totally ordered messages are delivered in the opposite order to the physical time at which they were sent. In fact, the definition of total

**Figure 11.13** Display from bulletin board program

Bulletin board: <i>os.interesting</i>		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

ordering allows message delivery to be ordered arbitrarily, as long as the order is the same at different processes. Since total ordering is not necessarily also a FIFO or causal ordering, we define the hybrid of *FIFO-total* ordering as one for which message delivery obeys both FIFO and total ordering; similarly, under *causal-total* ordering message delivery obeys both causal and total ordering.

The definitions of ordered multicast do not assume or imply reliability. For example, the reader should check that, under total ordering, if correct process  $p$  delivers message  $m$  and then delivers  $m'$ , then a correct process  $q$  can deliver  $m$  without also delivering  $m'$  or any other message ordered after  $m$ .

We can also form hybrids of ordered and reliable protocols. A reliable totally ordered multicast is often referred to in the literature as an *atomic multicast*. Similarly, we may form reliable FIFO multicast, reliable causal multicast and reliable versions of the hybrid ordered multicasts.

Ordering the delivery of multicast messages, as we shall see, can be expensive in terms of delivery latency and bandwidth consumption. The ordering semantics that we have described may delay the delivery of messages unnecessarily. That is, at the application level, a message may be delayed for another message that it does not in fact depend upon. For this reason, some have proposed multicast systems that use the application-specific message semantics alone to determine the order of message delivery [Cheriton and Skeen 1993, Pedone and Schiper 1999].

**The example of the bulletin board** ♦ To make multicast delivery semantics more concrete, consider an application in which users post messages to bulletin boards. Each user runs a bulletin-board application process. Every topic of discussion has its own process group. When a user posts a message to a bulletin board, the application multicasts the user's posting to the corresponding group. Each user's process is a member of the group for the topic in which he or she is interested, so that the user will receive just the postings concerning that topic.

Reliable multicast is required if every user is to receive every posting eventually. The users also have ordering requirements. Figure 11.13 shows the postings as they appear to a particular user. At a minimum, FIFO ordering is desirable, since then every posting from a given user – ‘A.Hanlon’, say – will be received in the same order, and users can talk consistently about A.Hanlon’s second posting.

Note that the message whose subjects are ‘Re: Microkernels’ (25) and ‘Re: Mach’ (27) appear after the messages to which they refer. A causally ordered multicast is needed to guarantee this relationship. Otherwise, arbitrary message delays could mean that, say, a message ‘Re: Mach’ could appear before the original message about Mach.

If the multicast delivery was totally ordered, then the numbering in the left-hand column would be consistent between users. Users could refer unambiguously, for example, to ‘message 24’.

In practice, the USENET bulletin board system implements neither causal nor total ordering. The communication costs of achieving these orderings on a large scale outweighs their advantages.

**Implementing FIFO ordering** ◊ FIFO-ordered multicast (with operations *FO-multicast* and *FO-deliver*) is achieved with sequence numbers, much as we would achieve it for one-to-one communication. We shall consider only non-overlapping groups. The reader should verify that the reliable multicast protocol that we defined on top of IP multicast in Section 11.4.2 also guarantees FIFO ordering, but we shall show how to construct a FIFO-ordered multicast on top of any given basic multicast. We use the variables  $S_g^p$  and  $R_g^q$  held at process  $p$  from the reliable multicast protocol of Section 11.4.2:  $S_g^p$  is a count of how many messages  $p$  has sent to  $g$  and, for each  $q$ ,  $R_g^q$  is the sequence number of the latest message  $p$  has delivered from process  $q$  that was sent to group  $g$ .

For  $p$  to *FO-multicast* a message to group  $g$ , it piggy backs the value  $S_g^p$  onto the message, *B-multicasts* the message to  $g$  and then increments  $S_g^p$  by 1. Upon receipt of a message from  $q$  bearing the sequence number  $S$ ,  $p$  checks whether  $S = R_g^q + 1$ . If so, this message is the next one expected from the sender  $q$  and  $p$  *FO-delivers* it, setting  $R_g^q := S$ . If  $S > R_g^q + 1$ , it places the message in the hold-back queue until the intervening messages have been delivered and  $S = R_g^q + 1$ .

Since all messages from a given sender are delivered in the same sequence, and since a message’s delivery is delayed until its sequence number has been reached, the condition for FIFO ordering is clearly satisfied. But this is so only under the assumption that groups are non-overlapping.

Note that we can use any implementation of *B-multicast* in this protocol. Moreover, if we use a reliable *R-multicast* primitive instead of *B-multicast*, then we obtain a reliable FIFO multicast.

**Implementing total ordering** ◊ The basic approach to implementing total ordering is to assign totally ordered identifiers to multicast messages so that each process makes the same ordering decision based upon these identifiers. The delivery algorithm is very similar to the one we described for FIFO ordering; the difference is that processes keep group-specific sequence numbers rather than process-specific sequence numbers. We only consider how to totally order messages sent to non-overlapping groups. We call the multicast operations *TO-multicast* and *TO-deliver*.

We discuss two main methods for assigning identifiers to messages. The first of these is for a process called a *sequencer* to assign them (Figure 11.14). A process wishing to *TO-multicast* a message  $m$  to group  $g$  attaches a unique identifier  $id(m)$  to it. The messages for  $g$  are sent to the sequencer for  $g$ ,  $sequencer(g)$ , as well as to the members of  $g$ . (The sequencer may be chosen to be a member of  $g$ .) The process  $sequencer(g)$  maintains a group-specific sequence number  $s_g$ , which it uses to assign increasing and consecutive sequence numbers to the messages that it *B-delivers*. It

**Figure 11.14** Total ordering using a sequencer1. Algorithm for group member  $p$ 

*On initialization:*  $r_g := 0;$

*To TO-multicast message  $m$  to group  $g$*

$B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle);$

*On  $B\text{-deliver}(\langle m, i \rangle)$  with  $g = \text{group}(m)$*

Place  $\langle m, i \rangle$  in hold-back queue;

*On  $B\text{-deliver}(m_{\text{order}} = \langle \text{"order"}, i, S \rangle)$  with  $g = \text{group}(m_{\text{order}})$*

wait until  $\langle m, i \rangle$  in hold-back queue and  $S = r_g$ ;

$TO\text{-deliver } m; // (\text{after deleting it from the hold-back queue})$

$r_g := S + 1;$

2. Algorithm for sequencer of  $g$ 

*On initialization:*  $s_g := 0;$

*On  $B\text{-deliver}(\langle m, i \rangle)$  with  $g = \text{group}(m)$*

$B\text{-multicast}(g, \langle \text{"order"}, i, s_g \rangle);$

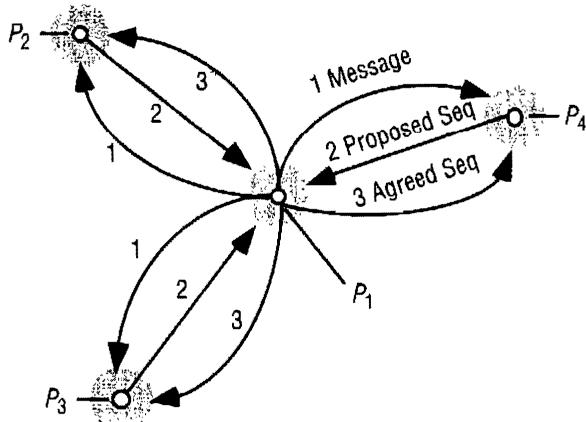
$s_g := s_g + 1;$

announces the sequence numbers by  $B\text{-multicasting}$  *order* messages to  $g$  (see Figure 11.14 for the details).

A message will remain in the hold-back queue indefinitely until it can be *TO-delivered* according to the corresponding sequence number. Since the sequence numbers are well defined (by the sequencer), the criterion for total ordering is met. Furthermore, if the processes use a FIFO-ordered variant of  $B\text{-multicast}$ , then the totally ordered multicast is also causally ordered. We leave the reader to show this.

The obvious problem with a sequencer-based scheme is that the sequencer may become a bottleneck and is a critical point of failure. Practical algorithms exist that address the problem of failure. Chang and Maxemchuk [1984] first suggested a multicast protocol employing a sequencer (which they called a *token site*). Kaashoek *et al.* [1989] developed a sequencer-based protocol for the Amoeba system. These protocols ensure that a message is in the hold-back queue at  $f + 1$  nodes before it is delivered; up to  $f$  failures can thus be tolerated. Like Chang and Maxemchuk, Birman *et al.* [1991] also employ a token-holding site that acts as a sequencer. The token can be passed from process to process so that, for example, if only one process sends totally-ordered multicasts then that process can act as the sequencer, saving communication.

The protocol of Kaashoek *et al.* uses hardware-based multicast – available on an Ethernet, for example – rather than reliable point-to-point communication. In the simplest variant of their protocol, processes send the message to be multicast to the sequencer, one-to-one. The sequencer multicasts the message itself, as well as the identifier and sequence number. This has the advantage that the other members of the

**Figure 11.15** The ISIS algorithm for total ordering

group receive only one message per multicast; its disadvantage is increased bandwidth utilization. The protocol is described in full at [www.cdk3.net/coordination](http://www.cdk3.net/coordination).

The second method that we examine for achieving totally ordered multicast is one in which the processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion. A simple algorithm – similar to one that was originally developed to implement totally ordered multicast delivery for the ISIS toolkit [Birman and Joseph 1987a] – is shown in Figure 11.15. Once more, a process  $B$ -*multicasts* its message to the members of the group. The group may be open or closed. The receiving processes propose sequence numbers for messages as they arrive and return these to the sender, which uses them to generate *agreed* sequence numbers.

Each process  $q$  in group  $g$  keeps  $A_g^q$ , the largest agreed sequence number it has observed so far for group  $g$ , and  $P_g^q$ , its own largest proposed sequence number. The algorithm for process  $p$  to multicast a message  $m$  to group  $g$  is as follows:

1.  $p$   $B$ -*multicasts*  $\langle m, i \rangle$  to  $g$ , where  $i$  is a unique identifier for  $m$ .
2. Each process  $q$  replies to the sender  $p$  with a proposal for the message's agreed sequence number of  $P_g^q := \text{Max}(A_g^q, P_g^q) + 1$ . In reality, we must include process identifiers in the proposed values  $P_g^q$  to ensure a total order, since otherwise different processes could propose the same integer value; but for the sake of simplicity we shall not make that explicit here. Each process provisionally assigns the proposed sequence number to the message and places it in its hold-back queue, which is ordered with the *smallest* sequence number at the front.
3.  $p$  collects all the proposed sequence numbers and selects the largest one  $a$  as the next agreed sequence number. It then  $B$ -*multicasts*  $\langle i, a \rangle$  to  $g$ . Each process  $q$  in  $g$  sets  $A_g^q := \text{Max}(A_g^q, a)$  and attaches  $a$  to the message (which is identified by  $i$ ). It reorders the message in the hold-back queue if the agreed sequence number differs from the proposed one. When the message at the front of the hold-back queue has been assigned its agreed sequence number, it is transferred to the tail of the delivery queue. Messages that have been assigned their agreed sequence

number but are not at the head of the hold-back queue are not yet transferred, however.

If every process agrees the same set of sequence numbers and delivers them in the corresponding order, then total ordering is satisfied. It is clear that correct processes ultimately agree on the same set of sequence numbers, but we must show that they are monotonically increasing and that no correct process can deliver a message prematurely.

Assume that a message  $m_1$  has been assigned an agreed sequence number and has reached the front of the hold-back queue. By construction, a message that is received after this stage will and should be delivered after  $m_1$ : it will have a larger proposed sequence number and thus a larger agreed sequence number than  $m_1$ . So let  $m_2$  be any other message that has not yet been assigned its agreed sequence number but which is on the same queue. We have that:

$$\text{agreedSequence}(m_2) \geq \text{proposedSequence}(m_2)$$

by the algorithm just given. Since  $m_1$  is at the front of the queue:

$$\text{proposedSequence}(m_2) > \text{agreedSequence}(m_1)$$

Therefore:

$$\text{agreedSequence}(m_2) > \text{agreedSequence}(m_1)$$

and total ordering is assured.

This algorithm has higher latency than the sequencer-based multicast: three messages are sent serially between the sender and the group before a message can be delivered.

Note that the total ordering chosen by this algorithm is not also guaranteed to be causally or FIFO-ordered: any two messages are delivered in an essentially arbitrary total order, influenced by communication delays.

For other approaches to implementing total ordering, see Melliar-Smith *et al.* [1990], Garcia-Molina and Spauster [1991] and Hadzilacos and Toueg [1994].

**Implementing causal ordering**  $\diamond$  We give an algorithm for non-overlapping closed groups based on that developed by Birman *et al.* [1991], shown in Figure 11.16, in which the causally-ordered multicast operations are *CO-multicast* and *CO-deliver*. The algorithm takes account of the happened-before relationship only as it is established by *multicast* messages. If the processes send one-to-one messages to one another, then these will not be accounted for.

Each process  $p_i$  ( $i = 1, 2, \dots, N$ ) maintains its own vector timestamp (see Section 10.4). The entries in the timestamp count the number of multicast messages from each process that happened-before the next message to be multicast.

To *CO-multicast* a message to group  $g$ , the process adds 1 to its entry in the timestamp, and *B-multcasts* the message along with its timestamp to  $g$ .

When a process  $p_i$  *B-delivers* a message from  $p_j$ , it must place it in the hold-back queue before it can *CO-deliver* it: until it is assured that it has delivered any messages that causally preceded it. To establish this,  $p_i$  waits until (a) it has delivered any earlier message sent by  $p_j$ , and (b) it has delivered any message that  $p_j$  had delivered at the

**Figure 11.16** Causal ordering using vector timestamps

Algorithm for group member  $p_i$  ( $i = 1, 2, \dots, N$ )

*On initialization*

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

*To CO-multicast message  $m$  to group  $g$*

$$V_i^g[i] := V_i^g[i] + 1;$$

$B\text{-multicast}(g, <V_i^g, m>);$

*On B-deliver(< $V_j^g, m$ >) from  $p_j$ , with  $g = \text{group}(m)$*

place  $<V_j^g, m>$  in hold-back queue;

wait until  $V_j^g[j] = V_i^g[j] + 1$  and  $V_j^g[k] \leq V_i^g[k]$  ( $k \neq j$ );

$CO\text{-deliver } m;$  // after removing it from the hold-back queue

$$V_i^g[j] := V_i^g[j] + 1;$$

time it multicasts the message. Both of those conditions can be detected by examining vector timestamps, as shown in Figure 11.16. Note that a process can immediately *CO-deliver* to itself any message that it *CO-multcasts*, although this is not described in Figure 11.16.

Each process updates its vector timestamp upon delivering any message, to maintain the count of causally precedent messages. It does this by incrementing the  $j$ th entry in its timestamp by one. This is an optimization of the *merge* operation that appears in the rules for updating vector clocks in Section 10.4. We can make the optimization in view of the delivery condition in the algorithm of Figure 11.16, which guarantees that only the  $j$ th entry will increase.

We outline the proof of the correctness of this algorithm as follows. Suppose that  $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$ . Let  $V$  and  $V'$  be the vector timestamps of  $m$  and  $m'$ , respectively. It is straightforward to prove inductively from the algorithm that  $V < V'$ . In particular, if process  $p_k$  multicasts  $m$ , then  $V[k] \leq V'[k]$ .

Consider what happens when some correct process  $p_i$  *B-delivers*  $m'$  (as opposed to *CO-delivering* it) without first *CO-delivering*  $m$ . By the algorithm,  $V_i[k]$  can increase only when  $p_i$  delivers a message from  $p_k$ , when it increases by 1. But  $p_i$  has not received  $m$ , and therefore  $V_i[k]$  cannot increase beyond  $V[k] - 1$ . It is therefore not possible for  $p_i$  to *CO-deliver*  $m'$ , since this would require that  $V_i[k] \geq V'[k]$ , and therefore that  $V_i[k] \geq V[k]$ .

The reader should check that if we substitute the reliable *R-multicast* primitive in place of *B-multicast*, then we obtain a multicast that is both reliable and causally ordered.

Furthermore, if we combine the protocol for causal multicast with the sequencer-based protocol for totally ordered delivery, then we obtain message delivery that is both total and causal. The sequencer delivers messages according to the causal order and multicasts the sequence numbers for the messages in the order in which it receives them. The processes in the destination group do not deliver a message until they have received an *order* message from the sequencer and the message is next in the delivery sequence.

Since the sequencer delivers message in causal order, and since all other processes deliver messages in the same order as the sequencer, the ordering is indeed both total and causal.

**Overlapping groups** ◊ We have considered only non-overlapping groups in the definitions and algorithms for FIFO, total and causal ordering semantics. This simplifies the problem but it is not satisfactory, since in general processes need to be members of multiple overlapping groups. For example, a process may be interested in events from multiple sources, and thus join a corresponding set of event-distribution groups.

We can extend the ordering definitions to global orders [Hadjilacos and Toueg 1994], in which we have to consider that if message  $m$  is multicast to  $g$ , and if message  $m'$  is multicast to  $g'$ , then both messages are addressed to the members of  $g \cap g'$ .

*Global FIFO ordering:* If a correct process issues  $\text{multicast}(g, m)$  and then  $\text{multicast}(g', m')$ , then every correct process in  $g \cap g'$  that delivers  $m'$  will deliver  $m$  before  $m'$ .

*Global causal ordering:* If  $\text{multicast}(g, m) \rightarrow \text{multicast}(g', m')$ , where  $\rightarrow$  is the happened-before relation induced by any chain of multicast messages, then any correct process in  $g \cap g'$  that delivers  $m'$  will deliver  $m$  before  $m'$ .

*Pairwise total ordering:* If a correct process delivers message  $m$  sent to  $g$  before it delivers  $m'$  sent to  $g'$ , then any other correct process in  $g \cap g'$  that delivers  $m'$  will deliver  $m$  before  $m'$ .

*Global total ordering:* Let ' $<$ ' be the relation of ordering between delivery events. We require that ' $<$ ' obeys pairwise total ordering and that it is acyclic – under pairwise total ordering, ' $<$ ' is not acyclic by default.

One way of implementing these orders would be to multicast each message  $m$  to the group of *all* processes in the system. Each process either discards or delivers the message according to whether it belongs to  $\text{group}(m)$ . This would be an inefficient and unsatisfactory implementation: a multicast should involve as few processes as possible beyond the members of the destination group. Alternatives are explored in Birman *et al.* [1991], Garcia-Molina and Spauster [1991], Hadjilacos and Toueg [1994], Kindberg [1995] and Rodrigues *et al.* [1998].

**Multicast in synchronous and asynchronous systems** ◊ In this section, we have described algorithms for reliable unordered multicast, (reliable) FIFO-ordered multicast, (reliable) causally ordered multicast and totally ordered multicast. We also indicated how to achieve a multicast that is both totally and causally ordered. We leave the reader to devise an algorithm for a multicast primitive that guarantees both FIFO and total ordering. All the algorithms that we have described work correctly in asynchronous systems.

We did not, however, give an algorithm that guarantees both reliable and totally ordered delivery. Surprising though it may seem, while possible in a *synchronous* system, a protocol with these guarantees is *impossible* in an asynchronous distributed system – even one that at worst suffered a single process crash failure. We return to this point in the next section.

## 11.5 Consensus and related problems

This section introduces the problem of consensus [Pease *et al.* 1980, Lamport *et al.* 1982] and the related problems of byzantine generals and interactive consistency. We shall refer to these collectively as problems of *agreement*. Roughly speaking, the problem is for processes to agree on a value after one or more of the processes has proposed what that value should be.

For example, in Chapter 2 we described a situation in which two armies should decide consistently to attack or retreat. Similarly, we may require that all the correct computers controlling a spaceship’s engines should decide ‘proceed’, or all of them decide ‘abort’, after each has proposed one action or the other. In a transaction to transfer funds from one account to another, the computers involved must consistently agree to perform the respective debit and credit. In mutual exclusion, the processes agree on which process can enter the critical section. In an election, the processes agree on which is the elected process. In totally ordered multicast, the processes agree on the order of message delivery.

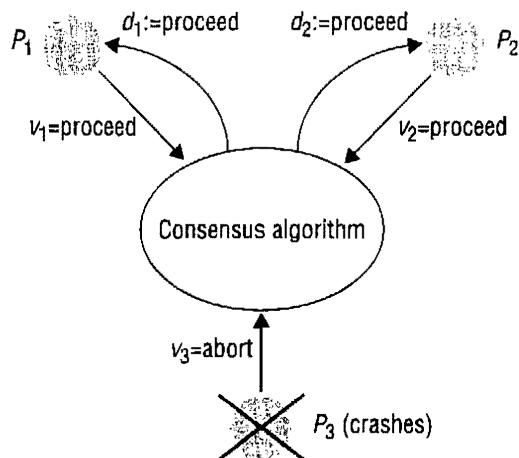
Protocols exist that are tailored to these individual types of agreement. We described some of them above, and Chapters 12 and 13 examine transactions. But it is useful for us to consider more general forms of agreement, in a search for common characteristics and solutions.

This section defines consensus more precisely and relates it to three related agreement problems: byzantine generals, interactive consistency and totally ordered multicast. We go on to examine under what circumstances the problems can be solved, and sketch some solutions. In particular, we shall discuss the well-known impossibility result of Fischer *et al.* [1985], which states that in an asynchronous system a collection of processes containing only one faulty process cannot be guaranteed to reach consensus. Finally, we consider how it is that practical algorithms exist despite the impossibility result.

### 11.5.1 System model and problem definitions

Our system model includes a collection of processes  $p_i$  ( $i = 1, 2, \dots, N$ ) communicating by message passing. An important requirement that applies in many practical situations is for consensus to be reached even in the presence of faults. We assume, as before, that communication is reliable but that processes may fail. In this section, we shall consider byzantine (arbitrary) process failures, as well as crash failures. We shall sometimes specify an assumption that up to some number  $f$  of the  $N$  processes are faulty – that is, they exhibit some specified types of fault; the remainder of the processes are correct.

If arbitrary failures can occur, then another factor in specifying our system is whether the processes digitally sign the messages that they send (see Section 7.4). If processes sign their messages, then a faulty process is limited in the harm it can do. Specifically, during an agreement algorithm it cannot make a false claim about the values that a correct process has sent to it. The relevance of message signing will become clearer when we discuss solutions to the byzantine generals problem. By default, we assume that signing does not take place.

**Figure 11.17** Consensus for three processes

**Definition of the consensus problem** ◊ To reach consensus, every process  $p_i$  begins in the *undecided* state and *proposes* a single value  $v_i$ , drawn from a set  $D$  ( $i = 1, 2, \dots, N$ ). The processes communicate with one another, exchanging values. Each process then sets the value of a *decision variable*  $d_i$ . In doing so it enters the *decided* state, in which it may no longer change  $d_i$  ( $i = 1, 2, \dots, N$ ). Figure 11.17 shows three processes engaged in a consensus algorithm. Two processes propose ‘proceed’ and a third proposes ‘abort’ but then crashes. The two processes that remain correct each decide ‘proceed’.

The requirements of a consensus algorithm are that the following conditions should hold for every execution of it:

*Termination:* Eventually each correct process sets its decision variable.

*Agreement:* The decision value of all correct processes is the same: if  $p_i$  and  $p_j$  are correct and have entered the *decided* state, then  $d_i = d_j$  ( $i, j = 1, 2, \dots, N$ ).

*Integrity:* If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value.

Variations on the definition of integrity may be appropriate, according to the application. For example, a weaker type of integrity would be for the decision value to equal a value that some correct process proposed – not necessarily all of them. We shall use the definition stated above.

To help in understanding how the formulation of the problem translates into an algorithm, consider a system in which processes cannot fail. It is then straightforward to solve consensus. For example, we can collect the processes into a group and have each process reliably multicast its proposed value to the members of the group. Each process waits until it has collected all  $N$  values (including its own). It then evaluates the function  $\text{majority}(v_1, v_1, \dots, v_N)$ , which returns the value that occurs most often among its arguments, or the special value  $\perp \notin D$  if no majority exists. Termination is guaranteed by the reliability of the multicast operation. Agreement and integrity are guaranteed by

the definition of *majority*, and the integrity property of a reliable multicast. Every process receives the same set of proposed values, and every process evaluates the same function of those values. So they must all agree, and if every process proposed the same value, then they all decide on this value.

Note that *majority* is only one possible function that the processes could use to agree upon a value from the candidate values. For example, if the values are ordered then the functions *minimum* and *maximum* may be appropriate.

If processes can crash then this introduces the complication of detecting failures, and it is not immediately clear that a run of the consensus algorithm can terminate. In fact, if the system is asynchronous then it may not; we shall return to this point shortly.

If processes can fail in *arbitrary* (byzantine) ways, then faulty processes can in principle communicate random values to the others. This may seem unlikely in practice, but it is not beyond the bounds of possibility for a process with a bug to fail in this way. Moreover, the fault may not be accidental but the result of mischievous or malevolent operation. Someone could deliberately make a process send different values to different peers in an attempt to thwart the others, which are trying to reach consensus. In case of inconsistency, correct processes must compare what they have received with what other processes claim to have received.

**The byzantine generals problem** ◊ In the informal statement of the *byzantine generals problem* [Lamport *et al.* 1982], three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, are to decide to attack or retreat. But one or more of the generals may be ‘treacherous’ – that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.

The byzantine generals problem differs from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value. The requirements are:

*Termination:* Eventually each correct process sets its decision variable.

*Agreement:* The decision value of all correct processes is the same: if  $p_i$  and  $p_j$  are correct and have entered the *decided* state, then  $d_i = d_j$  ( $i, j = 1, 2, \dots, N$ ).

*Integrity:* If the commander is correct, then all correct processes decide on the value that the commander proposed.

Note that, for the byzantine generals problem, integrity implies agreement when the commander is correct; but the commander need not be correct.

**Interactive consistency** ◊ The interactive consistency problem is another variant of consensus, in which every process proposes a single value. The goal of the algorithm is for the correct processes to agree on a *vector* of values, one for each process. We shall call this the ‘decision vector’. For example, the goal could be for each of a set of processes to obtain the same information about their respective states.

The requirements for interactive consistency are:

*Termination:* Eventually each correct process sets its decision variable.

*Agreement:* The decision vector of all correct processes is the same.

**Integrity:** If  $p_i$  is correct, then all correct processes decide on  $v_i$  as the  $i$ th component of their vector.

**Relating consensus to other problems** ◊ Although it is common to consider the byzantine generals problem with arbitrary process failures, in fact each of the three problems – consensus, byzantine generals and interactive consistency – is meaningful in the context of either arbitrary or crash failures. Similarly, each can be framed assuming either a synchronous or an asynchronous system.

It is sometimes possible to derive a solution to one problem using a solution to another. This is a very useful property, both because it increases our understanding of the problems and because by reusing solutions we can potentially save on implementation effort and complexity.

Suppose that there exist solutions to consensus (C), byzantine generals (BG) and interactive consistency (IC) as follows:

$C_i(v_1, v_2, \dots, v_N)$  returns the decision value of  $p_i$  in a run of the solution to the consensus problem, where  $v_1, v_2, \dots, v_N$  are the values that the processes proposed.

$BG_i(j, v)$  returns the decision value of  $p_i$  in a run of the solution to the byzantine generals problem, where  $p_j$ , the commander, proposes the value  $v$ .

$IC_i(v_1, v_2, \dots, v_N)[j]$  returns the  $j$ th value in the decision vector of  $p_i$  in a run of the solution to the interactive consistency problem, where  $v_1, v_2, \dots, v_N$  are the values that the processes proposed.

The definitions of  $C_i$ ,  $BG_i$  and  $IC_i$  assume that a faulty process proposes a single notional value, even though it may have given different proposed values to each of the other processes. This is only a convenience: the solutions will not rely on any such notional value.

It is possible to construct solutions out of the solutions to other problems. We give three examples:

**IC from BG:** We construct a solution to IC from BG by running BG  $N$  times, once with each process  $p_i$  ( $i, j = 1, 2, \dots, N$ ) acting as the commander:

$$IC_i(v_1, v_2, \dots, v_N)[j] = BG_i(j, v_j) \quad (i, j = 1, 2, \dots, N)$$

**C from IC:** We construct a solution to C from IC by running IC to produce a vector of values at each process, then applying an appropriate function on the vector's values to derive a single value:

$$C_i(v_1, \dots, v_N) = \text{majority}(IC_i(v_1, \dots, v_N)[1], \dots, IC_i(v_1, \dots, v_N)[N])$$

( $i = 1, 2, \dots, N$ ), where *majority* is as defined above.

**BG from C:** We construct a solution to BG from C as follows:

- The commander  $p_j$  sends its proposed value  $v$  to itself and each of the remaining processes;
- All processes run C with the values  $v_1, v_2, \dots, v_N$  that they receive ( $p_j$  may be faulty);

**Figure 11.18** Consensus in a synchronous system

---

Algorithm for process  $p_i \in g$ ; algorithm proceeds in  $f + 1$  rounds

*On initialization*

$$Values_i^1 := \{v_i\}; Values_i^0 = \{\};$$

*In round  $r$  ( $1 \leq r \leq f + 1$ )*

$$B\text{-multicast}(g, Values_i^r - Values_i^{r-1}); // \text{Send only values that have not been sent}$$

$$Values_i^{r+1} := Values_i^r;$$

*while* (in round  $r$ )

$$\{$$

*On  $B\text{-deliver}(V_j)$  from some  $p_j$*

$$Values_i^{r+1} := Values_i^{r+1} \cup V_j;$$

$$\}$$

*After ( $f + 1$ ) rounds*

$$\text{Assign } d_i = \min(Values_i^{f+1});$$


---

- They derive  $BG_i(j, v) = C_i(v_1, v_2, \dots, v_N)$  ( $i = 1, 2, \dots, N$ ).

The reader should check that the termination, agreement and integrity conditions are preserved in each case. Fischer [1983] relates the three problems in more detail.

Solving consensus is equivalent to solving reliable and totally ordered multicast: given a solution to one, we can solve the other. Implementing consensus with a reliable and totally ordered multicast operation *RTO-multicast* is straightforward. We collect all the processes into a group  $g$ . To achieve consensus, each process  $p_i$  performs *RTO-multicast*( $g, v_i$ ). Then each process  $p_i$  chooses  $d_i = m_i$ , where  $m_i$  is the *first* value that  $p_i$  *RTO-delivers*. The termination property follows from the reliability of the multicast. The agreement and integrity properties follow from the reliability and total ordering of multicast delivery. Chandra and Toueg [1996] demonstrate how reliable and totally ordered multicast can be derived from consensus.

### 11.5.2 Consensus in a synchronous system

This section describes an algorithm that uses only a basic multicast protocol to solve consensus in a synchronous system. The algorithm assumes that up to  $f$  of the  $N$  processes exhibit crash failures.

To reach consensus, each correct process collects proposed values from the other processes. The algorithm proceeds in  $f + 1$  rounds, in each of which the correct processes *B-multicast* the values between themselves. At most  $f$  processes may crash, by assumption. At worst, all  $f$  crashes occurred during the rounds, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived are in a position to agree.

The algorithm, shown in Figure 11.18, is based on that by Dolev and Strong [1983] and its presentation by Attiya and Welch [1998]. The variable  $Values_i^r$  holds the set of proposed values known to process  $p_i$  at the beginning of round  $r$ . Each process

multicasts the set of values that it has not sent in previous rounds. It then takes delivery of similar multicast messages from other processes and records any new values. Although this is not shown in Figure 11.18, the duration of a round is limited by setting a timeout based on the maximum time for a correct process to multicast a message. After  $f + 1$  rounds, each process chooses the minimum value it has received as its decision value.

Termination is obvious from the fact that the system is synchronous. To check the correctness of the algorithm, we must show that each process arrives at the same set of values at the end of the final round. Agreement and integrity will then follow, because the processes apply the *minimum* function to this set.

Assume, to the contrary, that two processes differ in their final set of values. Without loss of generality, some correct process  $p_i$  possesses a value  $v$  that another correct process  $p_j$  ( $i \neq j$ ) does not possess. The only explanation for  $p_i$  possessing a proposed value  $v$  at the end that  $p_j$  does not possess is that any third process,  $p_k$  say, that managed to send  $v$  to  $p_i$  crashed before  $v$  could be delivered to  $p_j$ . In turn, any process sending  $v$  in the previous round must have crashed, to explain why  $p_k$  possesses  $v$  in that round but  $p_j$  did not receive it. Proceeding in this way, we have to posit at least one crash in each of the preceding rounds. But we have assumed that at most  $f$  crashes can occur, and there are  $f + 1$  rounds. We have arrived at a contradiction.

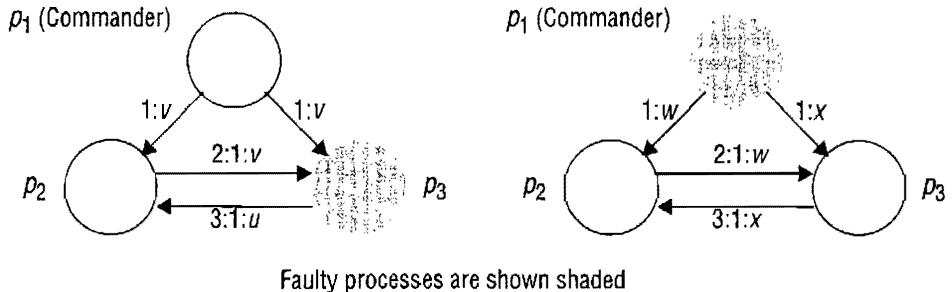
It turns out that *any* algorithm to reach consensus despite up to  $f$  crash failures requires at least  $f + 1$  rounds of message exchanges, no matter how it is constructed [Dolev and Strong 1983]. This lower bound also applies in the case of byzantine failures [Fischer and Lynch 1980].

### 11.5.3 The byzantine generals problem in a synchronous system

We discuss the byzantine generals problem in a synchronous system. Unlike the algorithm for consensus described in the previous section, here we assume that processes can exhibit arbitrary failures. That is, a faulty process may send any message with any value at any time; and it may omit to send any message. Up to  $f$  of the  $N$  processes may be faulty. Correct processes can detect the absence of a message through a timeout; but they cannot conclude that the sender has crashed, since it may be silent for some time and then send messages again.

We assume that the communication channels between pairs of processes are private. If a process could examine all the messages that other processes send, then it could detect the inconsistencies in what a faulty process sends to different processes. Our default assumption of channel reliability means that no faulty process can inject messages into the communication channel between correct processes.

Lamport *et al.* [1982] considered the case of three processes that send unsigned messages to one another. They showed that there is no solution that guarantees to meet the conditions of the byzantine generals problem if one process is allowed to fail. They generalized this result to show that no solution exists if  $N \leq 3f$ . We shall demonstrate these results shortly. They went on to give an algorithm that solves the byzantine generals problem in a synchronous system if  $N \geq 3f + 1$ , for unsigned (they call them ‘oral’) messages.

**Figure 11.19** Three byzantine generals

**Impossibility with three processes** ◊ Figure 11.19 shows two scenarios in which just one of three processes is faulty. In the left configuration one of the lieutenants,  $p_3$ , is faulty; on the right the commander,  $p_1$  is faulty. Each scenario in Figure 11.18 shows two rounds of messages: the values the commander sends, and the values that the lieutenants subsequently send to each other. The numeric prefixes serve to specify the sources of messages and to show the different rounds. Read the ‘:’ symbol in messages as ‘says’; for example, ‘3:1:u’ is the message ‘3 says 1 says u’.

In the left-hand scenario, the commander correctly sends the same value  $v$  to each of the other two processes, and  $p_2$  correctly echoes this to  $p_3$ . However,  $p_3$  sends a value  $u \neq v$  to  $p_2$ . All  $p_2$  knows at this stage is that it has received differing values; it cannot tell which were sent out by the commander.

In the right-hand scenario, the commander is faulty and sends differing values to the lieutenants. After  $p_3$  has correctly echoed the value  $x$  that it received,  $p_2$  is in the same situation as it was in when  $p_3$  was faulty: it has received two differing values.

If a solution exists, then process  $p_2$  is bound to decide on value  $v$  when the commander is correct, by the integrity condition. If we accept that no algorithm can possibly distinguish between the two scenarios,  $p_2$  must also choose the value sent by the commander in the right-hand scenario.

Following exactly the same reasoning for  $p_3$ , assuming that it is correct, we are forced to conclude, by symmetry, that  $p_3$  also chooses the value sent by the commander as its decision value. But this contradicts the agreement condition (the commander sends differing values if it is faulty). So no solution is possible.

Note that this argument rests on our intuition that nothing can be done to improve a correct general’s knowledge beyond the first stage, where it cannot tell which process is faulty. It is possible to prove the correctness of this intuition [Pease *et al.* 1980]. Byzantine agreement *can* be reached for three generals, with one of them faulty, if the generals digitally sign their messages.

**Impossibility with  $N \leq 3f$**  ◊ Pease *et al.* generalized the basic impossibility result for three processes, to prove that no solution is possible if  $N \leq 3f$ . In outline, the argument is as follows. Assume that a solution exists with  $N \leq 3f$ . Let each of three processes  $p_1$ ,  $p_2$  and  $p_3$  use the solution to simulate the behaviour of  $n_1$ ,  $n_2$  and  $n_3$  generals, respectively, where  $n_1 + n_2 + n_3 = N$  and  $n_1, n_2, n_3 \leq N/3$ . We assume, furthermore, that one of the three processes is faulty. Those of  $p_1$ ,  $p_2$  and  $p_3$  that are correct simulate correct generals: they simulate the interactions of their own generals internally and send

messages from their generals to those simulated by other processes. The faulty process's simulated generals are faulty: the messages that it sends as part of the simulation to the other two processes may be spurious. Since  $N \leq 3f$  and  $n_1, n_2, n_3 \leq N/3$ , at most  $f$  simulated generals are faulty.

Because the algorithm that the processes run is assumed to be correct, the simulation terminates. The correct simulated generals (in the two correct processes) agree and satisfy the integrity property. But now we have a means for the two correct processes out of the three to reach consensus: each decides on the value chosen by all of their simulated generals. This contradicts our impossibility result for three processes, with one faulty.

**Solution with one faulty process** ◊ There is not sufficient space to describe fully the algorithm of Pease *et al.* that solves the byzantine generals problem in a synchronous system with  $N \geq 3f + 1$ . Instead, we give the operation of the algorithm for the case  $N \geq 4$ ,  $f = 1$  and illustrate it for  $N = 4$ ,  $f = 1$ .

The correct generals reach agreement in two rounds of messages:

- In the first round, the commander sends a value to each of the lieutenants.
- In the second round, each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus  $N - 2$  values from its peers. If the commander is faulty, then all the lieutenants are correct and each will have gathered exactly the set of values that the commander sent out. Otherwise, one of the lieutenants is faulty; each of its correct peers receives  $N - 2$  copies of the value that the commander sent, plus a value that the faulty lieutenant sent to it.

In either case, the correct lieutenants need only apply a simple majority function to the set of values they receive. Since  $N \geq 4$ ,  $(N - 2) \geq 2$ . Therefore, the *majority* function will ignore any value that a faulty lieutenant sent, and it will produce the value that the commander sent if the commander is correct.

We now illustrate the algorithm that we have just outlined for the case of four generals. Figure 11.20 shows two scenarios similar to those in Figure 11.19, but in this case there are four processes, one of which is faulty. As in Figure 11.19, in the left-hand configuration one of the lieutenants,  $p_3$ , is faulty; on the right, the commander,  $p_1$ , is faulty.

In the left-hand case, the two correct lieutenant processes agree, deciding on the commander's value:

$$p_2 \text{ decides on } \text{majority}(v, u, v) = v$$

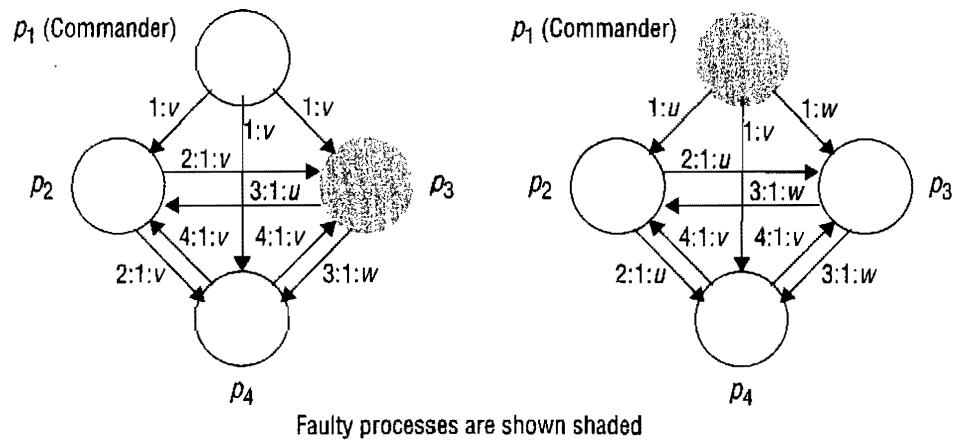
$$p_4 \text{ decides on } \text{majority}(v, v, w) = v$$

In the right-hand case the commander is faulty, but the three correct processes agree:

$p_2$ ,  $p_3$  and  $p_4$  decide on  $\text{majority}(u, v, w) = \perp$  (the special value  $\perp$  applies where no majority of values exists).

The algorithm takes account of the fact that a faulty process may omit to send a message. If a correct process does not receive a message within a suitable time limit (the system is synchronous), it proceeds as though the faulty process had sent it the value  $\perp$ .

**Discussion** ◊ We can measure the efficiency of a solution to the byzantine generals problem – or any other agreement problem – by asking:

**Figure 11.20** Four byzantine generals

- How many message rounds does it take? (This is a factor in how long it takes for the algorithm to terminate.)
- How many messages are sent, and of what size? (This measures the total bandwidth utilization and has an impact on the execution time.)

In the general case ( $f \geq 1$ ) the Lamport *et al.* algorithm for unsigned messages operates over  $f + 1$  rounds. In each round, a process sends to a subset of the other processes the values that it received in the previous round. The algorithm is very costly: it involves sending  $O(N^{f+1})$  messages.

Fischer and Lynch [1982] proved that any deterministic solution to consensus assuming byzantine failures (and hence to the byzantine generals problem, as Section 11.5.1 showed) will take at least  $f + 1$  message rounds. So no algorithm can operate faster in this respect than that of Lamport *et al.* But there have been improvements in the message complexity, for example Garay and Moses [1993].

Several algorithms, such as that of Dolev and Strong [1983], take advantage of signed messages. Dolev and Strong's algorithm again takes  $f + 1$  rounds, but the number of messages sent is only  $O(N^2)$ .

The complexity and cost of the solutions suggest that they are applicable only where the threat is great. If faulty hardware is the source of the threat, then the likelihood of truly arbitrary behaviour is small. Solutions that are based on more detailed knowledge of the fault model may be more efficient [Barborak *et al.* 1993]. If malicious users are the source of the threat, then a system to counter them is likely to use digital signatures; a solution without signatures is impractical.

#### 11.5.4 Impossibility in asynchronous systems

We have provided solutions to consensus and the byzantine generals problem (and hence, by derivation, to interactive consistency) in synchronous systems. However, all these solutions relied upon the system being synchronous. The algorithms assume that message exchanges take place in rounds, and that processes are entitled to timeout and

assume that a faulty process has not sent them a message within the round, because the maximum delay has been exceeded.

Fischer *et al.* [1985] proved that no algorithm can guarantee to reach consensus in an asynchronous system, even with one process crash failure. In an asynchronous system, processes can respond to messages at arbitrary times, so a crashed process is indistinguishable from a slow one. Their proof, which is beyond the scope of this book, involves showing that there is always some continuation of the processes' execution that avoids consensus being reached.

We immediately know from the result of Fischer *et al.* that there is no guaranteed solution in an asynchronous system to the byzantine generals problem, to interactive consistency or to totally ordered and reliable multicast. If there were such a solution then, by the results of Section 11.5.1, we would have a solution to consensus – contradicting the impossibility result.

Note the word 'guarantee' in the statement of the impossibility result. The result does not mean that processes can *never* reach distributed consensus in an asynchronous system if one is faulty. It allows that consensus can be reached with some probability greater than zero, confirming what we know in practice. For example, despite the fact that our systems are often effectively asynchronous, transaction systems have been reaching consensus regularly for many years.

One approach to working around the impossibility result is to consider *partially synchronous* systems, which are sufficiently weaker than synchronous systems to be useful as models of practical systems, and sufficiently stronger than asynchronous systems for consensus to be solvable in them [Dwork *et al.* 1988]. That approach is beyond the scope of this book. However, three other techniques for working around the impossibility result that we shall now outline are fault masking, and reaching consensus by exploiting failure detectors and by randomizing aspects of the processes' behaviour.

**Masking faults** ◊ The first technique is to avoid the impossibility result altogether by masking any process failures that occur (see Section 2.3.2 for an introduction to fault-masking). For example, transaction systems employ persistent storage, which survives crash failures. If a process crashes, then it is restarted (automatically, or by an administrator). The process places sufficient information in persistent storage at critical points in its program so that if it should crash and be restarted, it will find sufficient data to be able to continue correctly with its interrupted task. In other words, it will behave like a process that is correct, but which sometimes takes a long time to perform a processing step.

Of course, fault masking is generally applicable in system design. Chapter 13 discusses how transactional systems take advantage of persistent storage. Chapter 14 describes how process failures can also be masked by replicating software components.

**Consensus using failure detectors** ◊ Another method for circumventing the impossibility result is to employ failure detectors. Some practical systems employ 'perfect by design' failure detectors to reach consensus. No failure detector in an asynchronous system that works solely by message passing can really be perfect. However, processes can agree to *deem* a process that has not responded for more than a bounded time to have failed. An unresponsive process may not really have failed, but the remaining processes act as if it had done. They make the failure 'fail-silent' by discarding any subsequent messages that they do in fact receive from a 'failed' process.

In other words, we have effectively turned an asynchronous system into a synchronous one. This technique is used in the ISIS system [Birman 1993].

This method relies upon the failure detector usually being accurate. When it is inaccurate, then the system has to proceed without a group member that otherwise could potentially have contributed to the system's effectiveness. Unfortunately, making the failure detector reasonably accurate involves using long timeout values, forcing processes to wait a relatively long time (and not perform useful work) before concluding that a process has failed. Another issue that arises for this approach is network partitioning, which we discuss in Chapter 14.

A quite different approach is to use imperfect failure detectors, and to reach consensus while allowing suspected processes to behave correctly instead of excluding them. Chandra and Toueg [1996] analysed the properties that a failure detector must have in order to solve the consensus problem in an asynchronous system. They showed that consensus can be solved in an asynchronous system, even with an unreliable failure detector, if fewer than  $N/2$  processes crash and communication is reliable. The weakest type of failure detector for which this is so is called an *eventually weak failure detector*. This is one that is:

*Eventually weakly complete*: each faulty process is eventually suspected permanently by some correct process;

*Eventually weakly accurate*: after some point in time, at least one correct process is never suspected by any correct process.

Chandra and Toueg show that we cannot implement an eventually weak failure detector in an asynchronous system by message passing alone. However, we described a message-based failure detector in Section 11.1 that adapts its timeout values according to observed response times. If a process or the connection to it is very slow, then the timeout value will grow so that cases of falsely suspecting a process become rare. In the case of many real systems, this algorithm behaves sufficiently closely to an eventually weak failure detector for practical purposes.

Chandra and Toueg's consensus algorithm allows falsely suspected processes to continue their normal operations and allows processes that have suspected them to receive messages from them and process those messages normally. This makes the application programmer's life complicated, but it has the advantage that correct processes are not wasted by being falsely excluded. Moreover, timeouts for detecting failures can be set less conservatively than with the ISIS approach.

**Consensus using randomization** ♦ The result of Fischer *et al.* depends on what we can consider to be an 'adversary'. This is a 'character' (actually just a collection of random events) who can exploit the phenomena of asynchronous systems so as to foil the processes' attempts to reach consensus. The adversary manipulates the network to delay messages so that they arrive at just the wrong time, and similarly it slows down or speeds up the processes just enough so that they are in the 'wrong' state when they receive a message.

The third technique that addresses the impossibility result is to introduce an element of chance in the processes' behaviour, so that the adversary cannot exercise its thwarting strategy effectively. Consensus might still not be reached in some cases, but this method enables processes to reach consensus in a finite *expected* time. A

probabilistic algorithm that solves consensus even with byzantine failures can be found in Canetti and Rabin [1993].

## 11.6 Summary

The chapter began by discussing the need for processes to access shared resources under conditions of mutual exclusion. Locks are not always implemented by the servers that manage the shared resources, and a separate distributed mutual exclusion service is then required. Three algorithms were considered that achieve mutual exclusion: one employing a central server, a ring-based algorithm, and a multicast-based algorithm using logical clocks. None of these mechanisms can withstand failure as we described them, although they can be modified to tolerate some faults.

Then the chapter considered a ring-based algorithm and the bully algorithm, whose common aim is to elect a process uniquely from a given set – even if several elections take place concurrently. The Bully algorithm could be used, for example, to elect a new master time server, or a new lock server, when the previous one fails.

The chapter described multicast communication. It discussed reliable multicast, in which the correct processes agree on the set of messages to be delivered; and multicast with FIFO, causal and total delivery ordering. We gave algorithms for reliable multicast and for all three types of delivery ordering.

Finally, we described the three problems of consensus, byzantine generals and interactive consistency. We defined the conditions for their solution and we showed relationships between these problems – including the relationship between consensus and reliable, totally ordered multicast.

Solutions exist in a synchronous system, and we described some of them. In fact, solutions exist even when arbitrary failures are possible. We outlined part of the solution to the byzantine generals problem of Lamport *et al.* More recent algorithms have lower complexity, but in principle none can better the  $f + 1$  rounds taken by this algorithm, unless messages are digitally signed.

The chapter ended by describing the fundamental result of Fischer *et al.* concerning the impossibility of guaranteeing consensus in an asynchronous system. We discussed how it is that, nonetheless, systems regularly do reach agreement in asynchronous systems.

## EXERCISES

- 11.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel? *page 422*
- 11.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant? *page 425*
- 11.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay. *page 425*

- 
- 11.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order. *page 426*
- 11.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the resultant system is fault tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed? *page 426*
- 11.6 Give an example execution of the ring-based algorithm to show that processes are not necessarily granted entry to the critical section in happened-before order. *page 427*
- 11.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2? *page 429*
- 11.8 In the Bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm? *page 434*
- 11.9 Suggest how to adapt the Bully algorithm to deal with temporary network partitions (slow communication) and slow processes. *page 436*
- 11.10 Devise a protocol for basic multicast over IP multicast. *page 438*
- 11.11 How, if at all, should the definitions of integrity, agreement and validity for reliable multicast change for the case of open groups? *page 439*
- 11.12 Explain why reversing the order of the lines '*R-deliver m*' and '*if (q ≠ p) then B-multicast(g, m); end if*' in Figure 11.10 makes the algorithm no longer satisfy uniform agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform agreement? *page 440*
- 11.13 Explain whether the algorithm for reliable multicast over IP multicast works for open as well as closed groups. Given any algorithm for closed groups, how, simply, can we derive an algorithm for open groups? *page 440*
- 11.14 Consider how to address the impractical assumptions we made in order to meet the validity and agreement properties for the reliable multicast protocol based on IP multicast. Hint: add a rule for deleting retained messages when they have been delivered everywhere; and consider adding a dummy 'heartbeat' message, which is never delivered to the application, but which the protocol sends if the application has no message to send. *page 440*
- 11.15 Show that the FIFO-ordered multicast algorithm does not work for overlapping groups, by considering two messages sent from the same source to two overlapping groups, and considering a process in the intersection of those groups. Adapt the protocol to work for this case. Hint: processes should include with their messages the latest sequence numbers of messages sent to *all* groups. *page 445*
- 11.16 Show that, if the basic multicast that we use in the algorithm of Figure 11.14 is also FIFO-ordered, then the resultant totally-ordered multicast is also causally ordered. Is it the case that any multicast that is both FIFO-ordered and totally ordered is thereby causally ordered? *page 446*

- 11.17 Suggest how to adapt the causally ordered multicast protocol to handle overlapping groups. *page 449*
- 11.18 In discussing Maekawa's mutual exclusion algorithm, we gave an example of three subsets of a set of three processes that could lead to a deadlock. Use these subsets as multicast groups to show how a pairwise total ordering is not necessarily acyclic. *page 450*
- 11.19 Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem. *page 450*
- 11.20 We gave a solution to consensus from a solution to reliable and totally ordered multicast, which involved selecting the first value to be delivered. Explain from first principles why, in an asynchronous system, we could not instead derive a solution by using a reliable but not totally ordered multicast service and the 'majority' function. (Note that, if we could, then this would contradict the impossibility result of Fischer *et al.*!) Hint: consider slow/failed processes. *page 455*
- 11.21 Show that byzantine agreement can be reached for three generals, with one of them faulty, if the generals digitally sign their messages. *page 457*
- 11.22 Explain how to adapt the algorithm for reliable multicast over IP multicast to eliminate the hold-back queue – so that a received message that is not a duplicate can be delivered immediately, but without any ordering guarantees. Hint: use sets instead of sequence numbers to represent the messages that have been delivered so far. *page 441*



# DISTRIBUTED TRANSACTIONS

- 13.1 Introduction
- 13.2 Flat and nested distributed transactions
- 13.3 Atomic commit protocols
- 13.4 Concurrency control in distributed transactions
- 13.5 Distributed deadlocks
- 13.6 Transaction recovery
- 13.7 Summary

This chapter introduces distributed transactions – those that involve more than one server. Distributed transactions may be either flat or nested.

An atomic commit protocol is a cooperative procedure used by a set of servers involved in a distributed transaction. It enables the servers to reach a joint decision as to whether a transaction can be committed or aborted. This chapter describes the two-phase commit protocol, which is the most commonly used atomic commit protocol.

The section on concurrency control in distributed transactions discusses how locking, timestamp ordering and optimistic concurrency control may be extended for use with distributed transactions.

The use of locking schemes can lead to distributed deadlocks. Distributed deadlock detection algorithms are discussed.

Servers that provide transactions include a recovery manager whose concern is to ensure that the effects of transactions on the objects managed by a server can be recovered when it is replaced after a failure. The recovery manager saves the objects in permanent storage together with intentions lists and information about the status of each transaction.

## 13.1 Introduction

In Chapter 12, we discussed flat and nested transactions that accessed objects at a single server. In the general case, a transaction, whether flat or nested, will access objects located in several different computers. We use the term *distributed transaction* to refer to a flat or nested transaction that accesses objects managed by multiple servers.

When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved commit the transaction or all of them abort the transaction. To achieve this, one of the servers takes on a coordinator role, which involves ensuring the same outcome at all of the servers. The manner in which the coordinator achieves this depends on the protocol chosen. A protocol known as the ‘two-phase commit protocol’ is the most commonly used. This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

Concurrency control in distributed transactions is based on the methods discussed in Chapter 12. Each server applies local concurrency control to its own objects, which ensures that transactions are serialized locally. Distributed transactions must be serialized globally. How this is achieved varies as to whether locking, timestamp ordering or optimistic concurrency control is in use. In some cases, the transactions may be serialized at the individual servers, but at the same time a cycle of dependencies between the different servers may occur and a distributed deadlock arise.

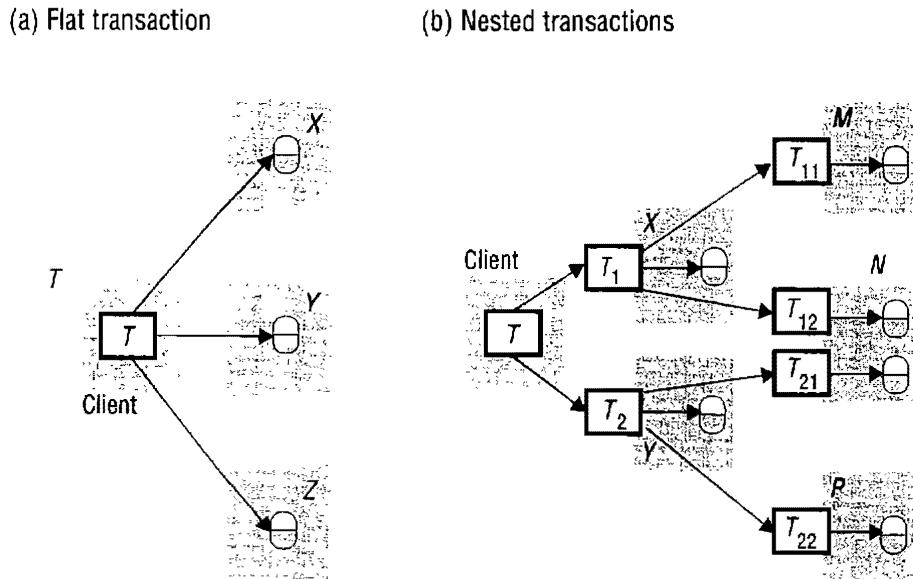
Transaction recovery is concerned with ensuring that all the objects involved in transactions are recoverable. In addition to that, it guarantees that the values of the objects reflect all the changes made by committed transactions and none of those made by aborted ones.

## 13.2 Flat and nested distributed transactions

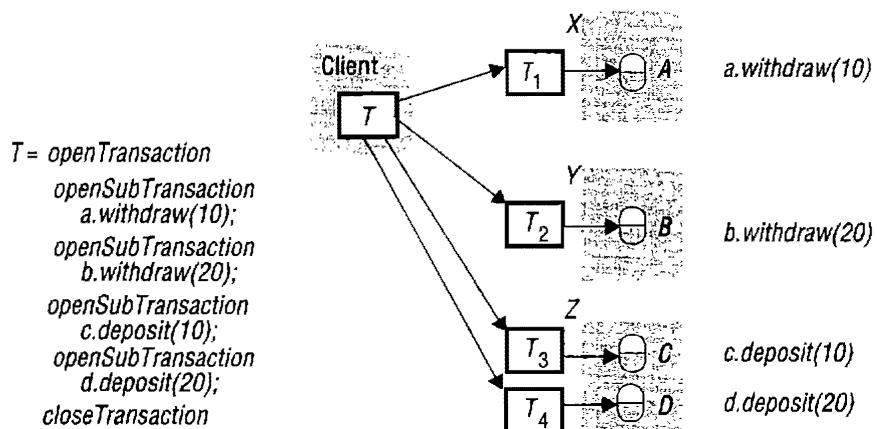
A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions.

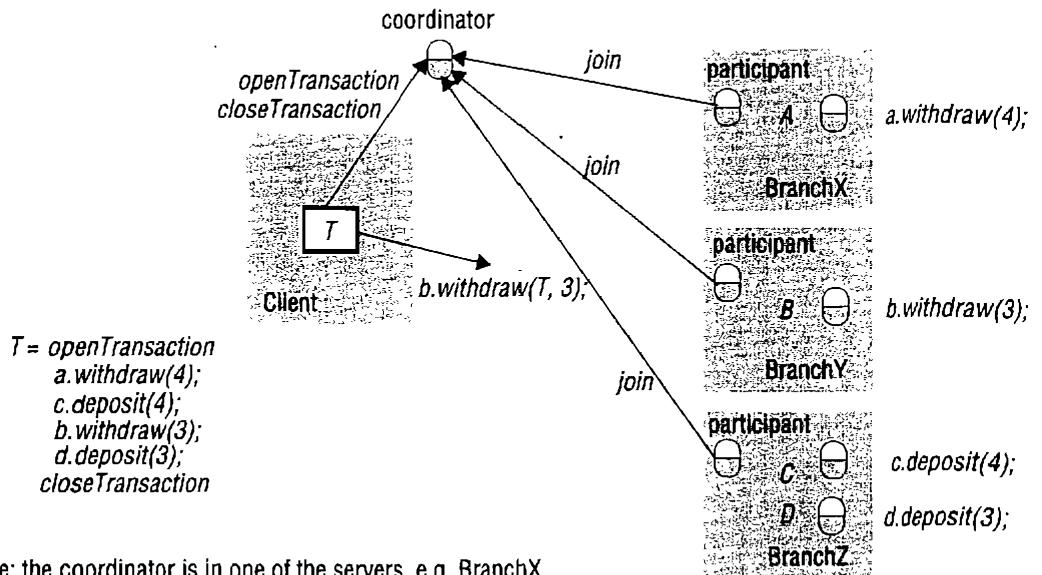
In a flat transaction, a client makes requests to more than one server. For example, in Figure 13.1(a), transaction  $T$  is a flat transaction that invokes operations on objects in servers  $X$ ,  $Y$  and  $Z$ . A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers’ objects sequentially. When servers use locking, a transaction can only be waiting for one object at a time.

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting. Figure 13.1(b) shows a client’s transaction  $T$  that opens two subtransactions  $T_1$  and  $T_2$ , which access objects at servers  $X$  and  $Y$ . The subtransactions  $T_1$  and  $T_2$  open further subtransactions  $T_{11}$ ,  $T_{12}$ ,  $T_{21}$  and  $T_{22}$ , which access objects at servers  $M$ ,  $N$  and  $P$ . In the nested case, subtransactions at the same level can run concurrently, so  $T_1$  and  $T_2$  are concurrent, and as they invoke objects in different servers, they can run in parallel. The four subtransactions  $T_{11}$ ,  $T_{12}$ ,  $T_{21}$  and  $T_{22}$  also run concurrently.

**Figure 13.1** Distributed transactions

Consider a distributed transaction in which a client transfers \$10 from account  $A$  to  $C$  and then transfers \$20 from  $B$  to  $D$ . Accounts  $A$  and  $B$  are at separate servers  $X$  and  $Y$  and accounts  $C$  and  $D$  are at server  $Z$ . If this transaction is structured as a set of four nested transactions, as shown in Figure 13.2, the four requests (two *deposit* and two *withdraw*) can run in parallel and the overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially.

**Figure 13.2** Nested banking transaction

**Figure 13.3** A distributed banking transaction

### 13.2.1 The coordinator of a distributed transaction

Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits. A client starts a transaction by sending an *openTransaction* request to a coordinator in any server, as described in Section 12.2. The coordinator that is contacted carries out the *openTransaction* and returns the resulting transaction identifier to the client. Transaction identifiers for distributed transactions must be unique within a distributed system. A simple way to achieve this is for a TID to contain two parts: the server identifier (for example, an IP address) of the server that created it and a number unique to the server.

The coordinator that opened the transaction becomes the *coordinator* for the distributed transaction and at the end is responsible for committing or aborting it. Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object we call the *participant*. Each participant is responsible for keeping track of all of the recoverable objects at that server involved in the transaction. The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator.

The interface for *Coordinator* shown in Figure 12.3 provides an additional method, *join*, which is used whenever a new participant joins the transaction:

---

*join(Trans, reference to participant)*

Informs a coordinator that a new participant has joined the transaction *Trans*.

---

The coordinator records the new participant in its participant list. The fact that the coordinator knows all the participants and each participant knows the coordinator will enable them to collect the information that will be needed at commit time.

Figure 13.3 shows a client whose (flat) banking transaction involves accounts *A*, *B*, *C* and *D* at servers BranchX, BranchY and BranchZ. The client's transaction, *T*, transfers \$4 from account *A* to account *C* and then transfers \$3 from account *B* to account *D*. The transaction described on the left is expanded to show that *openTransaction* and *closeTransaction* are directed to the coordinator, which would be situated in one of the servers involved in the transaction. Each server is shown with a *participant*, which joins the transaction by invoking the *join* method in the coordinator. When the client invokes one of the methods in the transaction, for example *b.withdraw(T, 3)*, the object receiving the invocation (*B* at BranchY in this case) informs its participant object that the object belongs to the transaction *T*. If it has not already informed the coordinator, the participant object uses the *join* operation to do so. In this example, we show the transaction identifier being passed as an additional argument so that the recipient can pass it on to the coordinator. By the time the client calls *closeTransaction*, the coordinator has references to all of the participants.

Note that it is possible for a participant to call *abortTransaction* in the coordinator if for some reason it is unable to continue with the transaction.

### 13.3 Atomic commit protocols

Transaction commit protocols were devised in the early 1970s, and the two-phase commit protocol appeared in Gray [1978]. The atomicity of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them. In the case of a distributed transaction, the client has requested the operations at more than one server. A transaction comes to an end when the client requests that a transaction be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out. This is an example of a *one-phase atomic commit protocol*.

This simple one-phase atomic commit protocol is inadequate because, in the case when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control. For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server. If optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. The coordinator may not know when a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.

The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted. In the first phase of the

protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a *prepared* state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared.

In the second phase of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction.

The problem is to ensure that all of the participants vote and that they all reach the same decision. This is fairly simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost or servers are temporarily unable to communicate with one another.

**Failure model for the commit protocols** ◊ Section 12.1.2 presents a failure model for transactions that applies equally to the two-phase (or any other) commit protocol. Commit protocols are designed to work in an asynchronous system in which servers may crash and messages may be lost. It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages. There are no byzantine faults – servers either crash or else they obey the messages they are sent.

The two-phase commit protocol is an example of a protocol for reaching a consensus. Chapter 11 asserts that consensus cannot be reached in an asynchronous system if processes sometimes fail. However, the two-phase commit protocol does reach consensus under those conditions. This is because crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.

### 13.3.1 The two-phase commit protocol

During the progress of a transaction, there is no communication between the coordinator and the participants apart from the participants informing the coordinator when they join the transaction. A client's request to commit (or abort) a transaction is directed to the coordinator. If the client requests *abortTransaction*, or if the transaction is aborted by one of the participants, the coordinator informs the participants immediately. It is when the client asks the coordinator to commit the transaction that two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; and in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes and its status in permanent storage – and is prepared to commit. The coordinator in a distributed transaction communicates with the participants to carry out the two-phase commit protocol by means of the operations summarized in Figure 13.4. The methods *canCommit*, *doCommit* and *doAbort* are

**Figure 13.4** Operations for two-phase commit protocol

*canCommit?(trans) → Yes / No*

Call from coordinator to participant to ask whether it can commit a transaction.  
Participant replies with its vote.

*doCommit(trans)*

Call from coordinator to participant to tell participant to commit its part of a transaction.

*doAbort(trans)*

Call from coordinator to participant to tell participant to abort its part of a transaction.

*haveCommitted(trans, participant)*

Call from participant to coordinator to confirm that it has committed the transaction.

*getDecision(trans) → Yes / No*

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

methods in the interface of the participant. The methods *haveCommitted* and *getDecision* are in the coordinator interface.

The two-phase commit protocol consists of a voting phase and a completion phase as shown in Figure 13.5. By the end of step (2) the coordinator and all the participants that voted *Yes* are prepared to commit. By the end of step (3) the transaction is effectively completed. At step (3a) the coordinator and the participants are committed, so the coordinator can report a decision to commit to the client. At (3b) the coordinator reports a decision to abort to the client.

At step (4) participants confirm that they have committed so that the coordinator knows when the information it has recorded about the transaction is no longer needed.

This apparently straightforward protocol could fail due to one or more of the servers crashing or due to a breakdown in communication between the servers. To deal with the possibility of crashing, each server saves information relating to the two-phase commit protocol in permanent storage. This information can be retrieved by a new process that is started to replace a crashed server. The recovery aspects of distributed transactions are discussed in Section 13.6.

The exchange of information between the coordinator and participants can fail when one of the servers crashes, or when messages are lost. Timeouts are used to avoid processes blocking for ever. When a timeout occurs at a process, it must take an appropriate action. To allow for this the protocol includes a timeout action for each step at which a process may block. These actions are designed to allow for the fact that in an asynchronous system, a timeout may not necessarily imply that a server has failed.

**Timeout actions in the two-phase commit protocol** ◊ There are various stages in the protocol at which the coordinator or a participant cannot progress its part of the protocol until it receives another request or reply from one of the others.

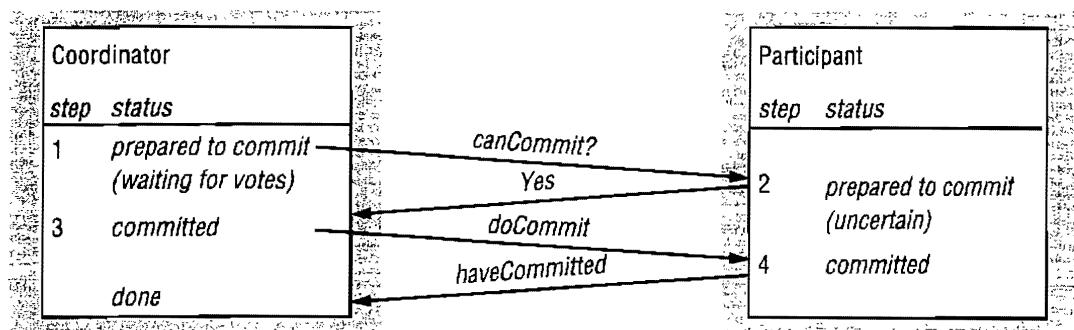
**Figure 13.5** The two-phase commit protocol*Phase 1 (voting phase):*

1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

*Phase 2 (completion according to outcome of vote):*

3. The coordinator collects the votes (including its own).
  - (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
  - (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Consider first the situation where a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort the transaction. See step (2) in Figure 13.6. Such a participant is *uncertain* of the outcome and cannot proceed any further until it gets the outcome of the vote from the coordinator. The participant cannot decide unilaterally what to do next, and meanwhile the objects used by its transaction cannot be released for use by other transactions. The participant makes a *getDecision* request to the coordinator to determine the outcome of the transaction. When it gets the reply it continues the protocol at step (4) in Figure 13.5. If

**Figure 13.6** Communication in two-phase commit protocol

the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the *uncertain* state.

Other alternative strategies are available for the participants to obtain a decision cooperatively instead of contacting the coordinator. These strategies have the advantage that they may be used when the coordinator has failed. See Exercise 13.5 and Bernstein *et al.* [1987] for details. However, even with a cooperative protocol, if all the participants are in the *uncertain* state, they will be unable to get a decision until the coordinator or a participant with the knowledge is available.

Another point at which a participant may be delayed is when it has carried out all its client requests in the transaction but has not yet received a *canCommit?* call from the coordinator. As the client sends the *closeTransaction* to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time, for example by a timeout period on a lock. As no decision has been made at this stage, the participant can decide to *abort* unilaterally after some period of time.

The coordinator may be delayed when it is waiting for votes from the participants. As it has not yet decided the fate of the transaction it may decide to *abort* the transaction after some period of time. It must then announce *doAbort* to the participants who have already sent their votes. Some tardy participants may try to vote *Yes* after this, but their votes will be ignored and they will enter the *uncertain* state as described above.

**Performance of the two-phase commit protocol** ◊ Provided that all goes well – that is, that the coordinator and participants and the communication between them do not fail, the two-phase commit protocol involving  $N$  participants can be completed with  $N$  *canCommit?* messages and replies, followed by  $N$  *doCommit* messages. That is, the cost in messages is proportional to  $3N$ , and the cost in time is three rounds of messages. The *haveCommitted* messages are not counted in the estimated cost of the protocol, which can function correctly without them – their role is to enable servers to delete stale coordinator information.

In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol. However, the protocol is designed to tolerate a succession of failures (server crashes or lost messages) and is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed.

As noted in the section on timeouts, the two-phase commit protocol can cause considerable delays to participants in the *uncertain* state. These delays occur when the coordinator has failed and cannot reply to *getDecision* requests from participants. Even if a cooperative protocol allows participants to make *getDecision* requests to other participants, delays will occur if all the active participants are *uncertain*.

Three-phase commit protocols have been designed to alleviate such delays. They are more expensive in the number of messages and the number of rounds required for the normal (failure-free) case. For a description of three-phase commit protocols, see Exercise 13.2 and Bernstein *et al.* [1987].

**Figure 13.7** Operations in coordinator for nested transactions

*openSubTransaction(trans) → subTrans*

Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

*getStatus(trans) → committed, aborted, provisional*

Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed, aborted, provisional*.

### 13.3.2 Two-phase commit protocol for nested transactions

The outermost transaction in a set of nested transactions is called the *top-level transaction*. Transactions other than the top-level transaction are called *subtransactions*. In Figure 13.1(b), *T* is the top-level transaction, *T<sub>1</sub>*, *T<sub>2</sub>*, *T<sub>11</sub>*, *T<sub>12</sub>*, *T<sub>21</sub>* and *T<sub>22</sub>* are subtransactions. *T<sub>1</sub>* and *T<sub>2</sub>* are child transactions of *T*, which is referred to as their parent. Similarly, *T<sub>11</sub>* and *T<sub>12</sub>* are child transactions of *T<sub>1</sub>*, and *T<sub>21</sub>* and *T<sub>22</sub>* are child transactions of *T<sub>2</sub>*. Each subtransaction starts after its parent and finishes before it. Thus, for example, *T<sub>11</sub>* and *T<sub>12</sub>* start after *T<sub>1</sub>* and finish before it.

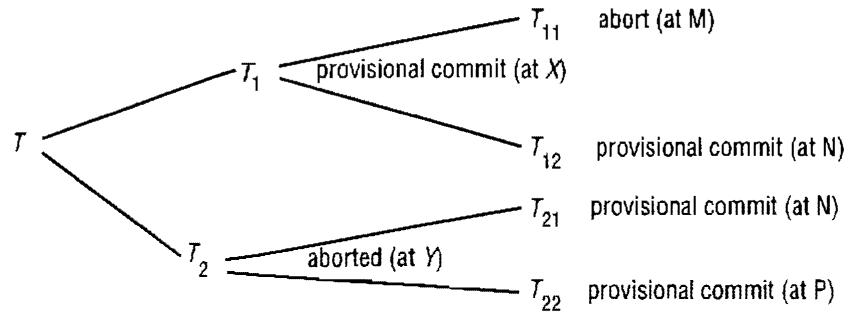
When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is not the same as being prepared: it is just a local decision and is not backed up on permanent storage. If the server crashes subsequently, its replacement will not be able to carry out a provisional commit. For this reason, a two-phase commit protocol is required for nested transactions to allow servers of provisionally committed transactions that have failed to abort them.

A coordinator for a subtransaction will provide an operation to open a subtransaction, together with an operation enabling the coordinator of a subtransaction to enquire whether its parent has yet committed or aborted, as shown in Figure 13.7.

A client starts a set of nested transactions by opening a top-level transaction with an *openTransaction* operation, which returns a transaction identifier for the top-level transaction. The client starts a subtransaction by invoking the *openSubTransaction* operation, whose argument specifies its parent transaction. The new subtransaction automatically *joins* the parent transaction, and a transaction identifier for a subtransaction is returned.

An identifier for a subtransaction must be an extension of its parent's TID, constructed in such a way that the identifier of the parent or top-level transaction of a subtransaction can be determined from its own transaction identifier. In addition, all subtransaction identifiers should be globally unique. The client makes a set of nested transactions come to completion by invoking *closeTransaction* or *abortTransaction* on the coordinator of the top-level transaction.

Meanwhile, each of the nested transactions carries out its operations. When they are finished, the server managing a subtransaction records information as to whether the subtransaction committed provisionally or aborted. Note that if its parent aborts, then the subtransaction will be forced to abort too.

**Figure 13.8** Transaction  $T$  decides whether to commit

Recall from Chapter 12 that a parent transaction – including a top-level transaction – can commit even if one of its child subtransactions has aborted. In such cases, the parent transaction will be programmed to take different actions according to whether a subtransaction has committed or aborted. For example, consider a banking transaction that is designed to perform all the ‘standing orders’ at a branch on a particular day. This transaction is expressed as several nested *Transfer* subtransactions, each of which consists of nested *deposit* and *withdraw* subtransactions. We assume that when an account is overdrawn, *withdraw* aborts and then the corresponding *Transfer* aborts. But there is no need to abort all the standing orders just because one *Transfer* subtransaction aborts. Instead of aborting, the top-level transaction will note the *Transfer* subtransactions that aborted and take appropriate actions.

Consider the top-level transaction  $T$  and its subtransactions shown in Figure 13.8, which is based on Figure 13.1(b). Each subtransaction has either provisionally committed or aborted. For example,  $T_{12}$  has provisionally committed and  $T_{11}$  has aborted, but the fate of  $T_{12}$  depends on its parent  $T_1$  and eventually on the top-level transaction,  $T$ . Although  $T_{21}$  and  $T_{22}$  have both provisionally committed,  $T_2$  has aborted and this means that  $T_{21}$  and  $T_{22}$  must also abort. Suppose that  $T$  decides to commit in spite of the fact that  $T_2$  has aborted, also that  $T_1$  decides to commit in spite of the fact that  $T_{11}$  has aborted.

When a top-level transaction completes, its coordinator carries out a two-phase commit protocol. The only reason for a participant subtransaction being unable to complete is if it has crashed since it completed its provisional commit. Recall that when each subtransaction was created, it *joined* its parent transaction. Therefore, the coordinator of each parent transaction has a list of its child subtransactions. When a nested transaction provisionally commits, it reports its status and the status of its descendants to its parent. When a nested transaction aborts, it just reports abort to its parent without giving any information about its descendants. Eventually, the top-level transaction receives a list of all the subtransactions in the tree, together with the status of each. Descendants of aborted subtransactions are actually omitted from this list.

The information held by each coordinator in the example shown in Figure 13.8 is shown in Figure 13.9. Note that  $T_{12}$  and  $T_{21}$  share a coordinator as they both run at server N. When subtransaction  $T_2$  aborted, it reported the fact to its parent,  $T$ , but without passing on any information about its subtransactions  $T_{21}$  and  $T_{22}$ . A subtransaction is an *orphan* if one of its ancestors aborts, either explicitly or because its coordinator crashed.

**Figure 13.9** Information held by coordinators of nested transactions:

<i>Coordinator of transaction</i>	<i>Child transactions</i>	<i>Participant</i>	<i>Provisional commit list</i>	<i>Abort list</i>
$T$	$T_1, T_2$	yes	$T_1, T_{12}$	$T_{11}, T_2$
$T_1$	$T_{11}, T_{12}$	yes	$T_1, T_{12}$	$T_{11}$
$T_2$	$T_{21}, T_{22}$	no (aborted)		$T_2$
$T_{11}$		no (aborted)		$T_{11}$
$T_{12}, T_{21}$		$T_{12}$ but not $T_{21}$	$T_{21}, T_{12}$	
$T_{22}$		no (parent aborted)	$T_{22}$	

In our example, subtransactions  $T_{21}$  and  $T_{22}$  are orphans because their parent aborted without passing information about them to the top-level transaction. Their coordinator can however, make enquiries about the status of their parent by using the *getStatus* operation. A provisionally committed subtransaction of an aborted transaction should be aborted, irrespective of whether the top-level transaction eventually commits.

The top-level transaction plays the role of coordinator in the two-phase commit protocol, and the participant list consists of the coordinators of all the subtransactions in the tree that have provisionally committed but do not have aborted ancestors. By this stage, the logic of the program has determined that the top-level transaction should try to commit whatever is left, in spite of some aborted subtransactions. In Figure 13.8, the coordinators of  $T$ ,  $T_1$  and  $T_{12}$  are participants and will be asked to vote on the outcome. If they vote to commit, then they must *prepare* their transactions by saving the state of the objects in permanent storage. This state is recorded as belonging to the top-level transaction of which it will form a part. The two-phase commit protocol may be performed in either a hierachic manner or in a flat manner.

The second phase of the two-phase commit protocol is the same as for the non-nested case. The coordinator collects the votes and then informs the participants as to the outcome. When it is complete, coordinator and participants will have committed or aborted their transactions.

**Hierachic two-phase commit protocol** ◊ In this approach, the two-phase commit protocol becomes a multi-level nested protocol. The coordinator of the top-level transaction communicates with the coordinators of the subtransactions for which it is the immediate parent. It sends *canCommit?* messages to each of the latter, which in turn pass them on to the coordinators of their child transactions (and so on down the tree). Each participant collects the replies from its descendants before replying to its parent. In our example,  $T$  sends *canCommit?* messages to the coordinator of  $T_1$  and then  $T_1$  sends *canCommit?* messages to  $T_{12}$  asking about descendants of  $T_1$ . The protocol does not include the coordinators of transactions such as  $T_2$ , which has aborted. Figure 13.10 shows the arguments required for *canCommit?* The first argument is the TID of the top-level transaction, for use when preparing the data. The second argument is the TID of the participant making the *canCommit?* call. The participant receiving the call looks in its transaction list for any provisionally committed transaction or subtransaction

**Figure 13.10** *canCommit?* for hierachic two-phase commit protocol

*canCommit? (trans, subTrans) → Yes / No*

Call a coordinator to ask coordinator of child subtransaction whether it can commit a subtransaction *subTrans*. The first argument *trans* is the transaction identifier of top-level transaction. Participant replies with its vote *Yes / No*.

---

**Figure 13.11** *canCommit?* for flat two-phase commit protocol

*canCommit? (trans, abortList) → Yes / No*

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote *Yes / No*.

---

matching the TID in the second argument. For example, the coordinator of  $T_{12}$  is also the coordinator of  $T_{21}$ , since they run in the same server, but when it receives the *canCommit?* call, the second argument will be  $T_1$  and it will deal only with  $T_{12}$ .

If a participant finds any subtransactions that match the second argument, it prepares the objects and replies with a *Yes* vote. If it fails to find any, then it must have crashed since it performed the subtransaction and it replies with a *No* vote.

**Flat two-phase commit protocol** ◊ In this approach, the coordinator of the top-level transaction sends *canCommit?* messages to the coordinators of all of the subtransactions in the provisional commit list. In our example, to the coordinators of  $T_1$  and  $T_{12}$ . During the commit protocol, the participants refer to the transaction by its top-level TID. Each participant looks in its transaction list for any transaction or subtransaction matching that TID. For example, the coordinator of  $T_{12}$  is also the coordinator of  $T_{21}$ , since they run in the same server ( $N$ ).

Unfortunately, this does not provide sufficient information to enable correct actions by participants such as the coordinator at server  $N$  that have a mix of provisionally committed and aborted subtransactions. If  $N$ 's coordinator is just asked to commit  $T$  it will end up by committing both  $T_{12}$  and  $T_{21}$ , because, according to its local information, both have provisionally committed. This is wrong in the case of  $T_{21}$ , because its parent,  $T_2$ , has aborted. To allow for such cases, the *canCommit?* operation for the flat commit protocol has a second argument that provides a list of aborted subtransactions, as shown in Figure 13.11. A participant can commit descendants of the top-level transaction unless they have aborted ancestors. When a participant receives a *canCommit?* request, it does the following:

- If the participant has any provisionally committed transactions that are descendants of the top-level transaction, *trans*:
  - check that they do not have aborted ancestors in the *abortList*. Then prepare to commit (by recording the transaction and its objects in permanent storage);

- those with aborted ancestors are aborted;
- send a *Yes* vote to the coordinator.
- If the participant does not have a provisionally committed descendent of the top-level transaction, it must have failed since it performed the subtransaction and it sends a *No* vote to the coordinator.

**A comparison of the two approaches** ◊ The hierachic protocol has the advantage that at each stage, the participant only need look for subtransactions of its immediate parent, whereas the flat protocol needs to have the abort list in order to eliminate transactions whose parents have aborted. Moss [1985] preferred the flat algorithm because it allows the coordinator of the top-level transaction to communicate directly with all of the participants, whereas the hierachic variant involves passing a series of messages down and up the tree in stages.

**Timeout actions** ◊ The two-phase commit protocol for nested transactions can cause the coordinator or a participant to be delayed at the same three steps as in the non-nested version. There is a fourth step at which subtransactions can be delayed. Consider provisionally committed child subtransactions of aborted subtransactions: they do not necessarily get informed of the outcome of the transaction. In our example,  $T_{22}$  is such a subtransaction – it has provisionally committed, but as its parent  $T_2$  has aborted, it does not become a participant. To deal with such situations, any subtransaction that has not received a *canCommit?* message will make an enquiry after a timeout period. The *getStatus* operation in Figure 13.7 allows a subtransaction to enquire whether its parent has committed or aborted. To make such enquiries possible, the coordinators of aborted subtransactions need to survive for a period. If an orphaned subtransaction cannot contact its parent, it will eventually abort.

## 13.4 Concurrency control in distributed transactions

Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions. Therefore, each server is responsible for applying concurrency control to its own objects. The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner.

This implies that if transaction  $T$  is before transaction  $U$  in their conflicting access to objects at one of the servers then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both  $T$  and  $U$ .

### 13.4.1 Locking

In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction. When locking is used for concurrency control, the objects remain locked and are unavailable

for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. Consider the following interleaving of transactions  $T$  and  $U$  at servers  $X$  and  $Y$ :

$T$			$U$		
$Write(A)$	at $X$	locks $A$			
$Read(B)$	at $Y$	waits for $U$	$Write(B)$	at $Y$	locks $B$
			$Read(A)$	at $X$	waits for $T$

The transaction  $T$  locks object  $A$  at server  $X$  and then transaction  $U$  locks object  $B$  at server  $Y$ . After that,  $T$  tries to access  $B$  at server  $Y$  and waits for  $U$ 's lock. Similarly, transaction  $U$  tries to access  $A$  at server  $X$  and has to wait for  $T$ 's lock. Therefore, we have  $T$  before  $U$  in one server and  $U$  before  $T$  in the other. These different orderings can lead to cyclic dependencies between transactions and a distributed deadlock situation arises. The detection and resolution of distributed deadlocks is discussed in the next section of this chapter. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

### 13.4.2 Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them. In distributed transactions, we require that each coordinator issue globally unique timestamps. A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. For example, if the version of an object accessed by transaction  $U$  commits after the version accessed by  $T$  at one server, then if  $T$  and  $U$  access the same object as one another at other servers, they must commit them in the same order. To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps. A timestamp consists of a pair  $\langle local\ timestamp, server-id \rangle$ . The agreed ordering of pairs of timestamps is based on a comparison in which the server-id part is less significant.

The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized. However, for reasons of efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with those issued by the other coordinators. When this is the case, the ordering of transactions generally

corresponds to the order in which they are started in real time. Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks (see Chapter 10).

When timestamp ordering is used for concurrency control, conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants. Therefore, any transaction that reaches the client request to commit should always be able to commit. Therefore, a participant in the two-phase commit protocol will normally agree to commit. The only situation in which a participant will not agree to commit is if it had crashed during the transaction.

### 13.4.3 Optimistic concurrency control

Recall that with optimistic concurrency control, each transaction is validated before it is allowed to commit. Transaction numbers are assigned at the start of validation and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers, each of which validates transactions that access its own objects. The validation at all of the servers takes place during the first phase of the two-phase commit protocol.

Consider the following interleavings of transactions  $T$  and  $U$ , which access objects  $A$  and  $B$  at servers  $X$  and  $Y$ , respectively.

$T$	$U$
$Read(A)$ at $X$	$Read(B)$ at $Y$
$Write(A)$	$Write(B)$
$Read(B)$ at $Y$	$Read(A)$ at $X$
$Write(B)$	$Write(A)$

The transactions access the objects in the order  $T$  before  $U$  at server  $X$  and in the order  $U$  before  $T$  at server  $Y$ . Now suppose that  $T$  and  $U$  start validation at about the same time, but server  $X$  validates  $T$  first and server  $Y$  validates  $U$  first. Recall that Section 12.5 recommends a simplification of the validation protocol that makes a rule that only one transaction may perform validation and update phases at a time. Therefore each server will be unable to validate the other transaction until the first one has completed. This is an example of commitment deadlock.

The validation rules in Section 12.5 assume that validation is fast, which is true for single-server transactions. However, in a distributed transaction, the two-phase commit protocol may take some time and will delay other transactions from entering validation until a decision on the current transaction has been obtained. In distributed optimistic transactions, each server applies a parallel validation protocol. This is an extension of either backward or forward validation to allow multiple transactions to be in the validation phase at the same time. In this extension, rule 3 must be checked as well as rule 2 for backward validation. That is, the write set of the transaction being validated must be checked for overlaps with the write set of earlier overlapping transactions. Kung and Robinson [1981] describe parallel validation in their paper.

If parallel validation is used, transactions will not suffer from commitment deadlock. However, if servers simply perform independent validations, it is possible that different servers of a distributed transaction may serialize the same set of transactions in different orders, for example with  $T$  before  $U$  at server  $X$  and  $U$  before  $T$  at server  $Y$  in our example.

The servers of distributed transactions must prevent this happening. One approach is that after a local validation by each server, a global validation is carried out [Ceri and Owicki 1982]. The global validation checks that the combination of the orderings at the individual servers is serializable; that is, that the transaction being validated is not involved in a cycle.

Another approach is that all of the servers of a particular transaction use the same globally unique transaction number at the start of the validation [Schlageter 1982]. The coordinator of the two-phase commit protocol is responsible for generating the globally unique transaction number and passes it to the participants in the *canCommit?* messages. As different servers may coordinate different transactions, the servers must (as in the distributed timestamp ordering protocol) have an agreed order for the transaction numbers they generate.

Agrawal *et al.* [1987] have proposed a variation of Kung and Robinson's algorithm that favours read-only transactions, together with an algorithm called MVGV (multi-version generalized validation). MVGV is a form of parallel validation that ensures that transaction numbers reflect serial order, but it requires that the visibility of some transactions be delayed after having committed. It also allows the transaction number to be changed so as to permit some transactions to validate that otherwise would have failed. The paper also proposes an algorithm for committing distributed transactions. It is similar to Schlageter's proposal in that a global transaction number has to be found. At the end of the read phase, the coordinator proposes a value for the global transaction number and each participant attempts to validate their local transactions using that number. However, if the proposed global transaction number is too small, some participants may not be able to validate their transaction and they negotiate with the coordinator for an increased number. If no suitable number can be found, then that participants will have to abort its transaction. Eventually, if all of the participants can validate their transactions the coordinator will have received proposals for transaction numbers from each of them. If common numbers can be found then the transaction will be committed.

## 13.5 Distributed deadlocks

The discussion of deadlocks in Section 12.4 shows that deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval, and transactions are aborted unnecessarily. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for

**Figure 13.12** Interleavings of transactions  $U$ ,  $V$  and  $W$ 

$U$	$V$	$W$
$d.deposit(10)$ lock $D$	$b.deposit(10)$ lock $B$	
$a.deposit(20)$ lock $A$ at $X$	at $Y$	$c.deposit(30)$ lock $C$ at $Z$
$b.withdraw(30)$ wait at $Y$	$c.withdraw(20)$ wait at $Z$	$a.withdraw(20)$ wait at $X$

graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-for graph.

Figure 13.12 shows the interleavings of the transactions  $U$ ,  $V$  and  $W$  involving the objects  $A$  and  $B$  managed by servers  $X$  and  $Y$  and objects  $C$  and  $D$  managed by server  $Z$ .

The complete wait-for graph in Figure 13.13(a) shows that a deadlock cycle consists of alternate edges, which represent a transaction waiting for an object and an object held by a transaction. As any transaction can only be waiting for one object at a time, objects can be left out of wait-for graphs, as shown in Figure 13.13(b).

Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions. Local wait-for graphs can be built by the lock manager at each server, as discussed in Chapter 12. In the above example, the local wait-for graphs of the servers are:

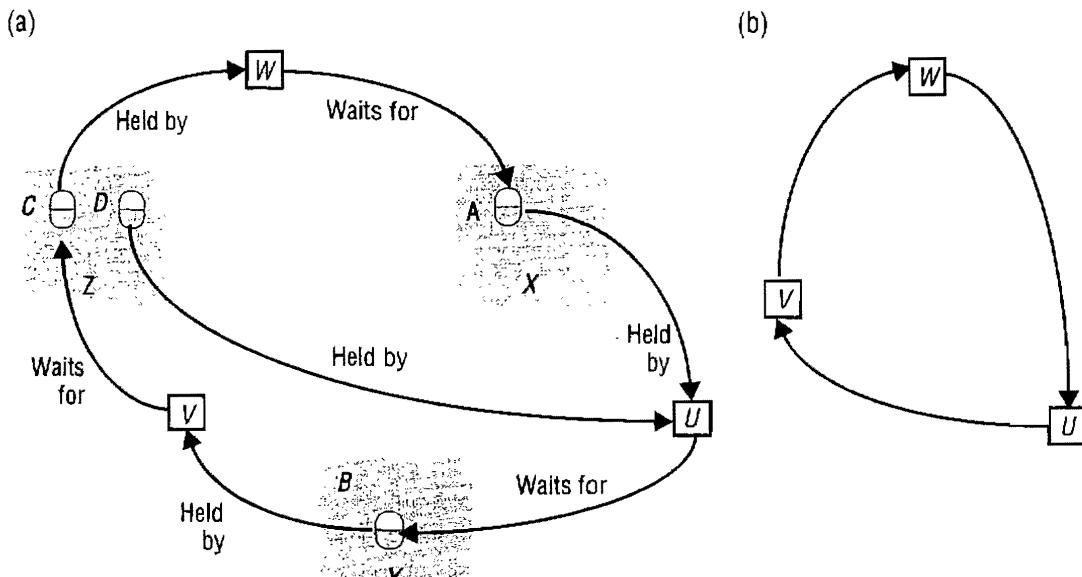
server  $Y$ :  $U \rightarrow V$  (added when  $U$  requests  $b.withdraw(30)$ )

server  $Z$ :  $V \rightarrow W$  (added when  $V$  requests  $c.withdraw(20)$ )

server  $X$ :  $W \rightarrow U$  (added when  $W$  requests  $a.withdraw(20)$ )

As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph.

**Figure 13.13** Distributed deadlock

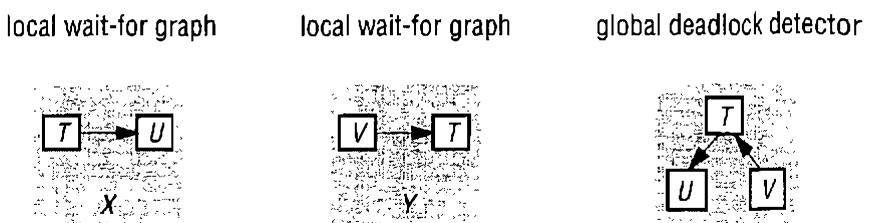
When it finds a cycle, it makes a decision on how to resolve the deadlock and informs the servers as to the transaction to be aborted to resolve the deadlock.

Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale. In addition, the cost of the frequent transmission of local wait-for graphs is high. If the global graph is collected less frequently, deadlocks may take longer to be detected.

**Phantom deadlocks** ◊ A deadlock that is ‘detected’ but is not really a deadlock is called a phantom deadlock. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Consider the case of a global deadlock detector that receives local wait-for graphs from servers  $X$  and  $Y$ , as shown in Figure 13.14. Suppose that transaction  $U$  then releases an object at server  $X$  and requests the one held by  $V$  at server  $Y$ . Suppose also that the global detector receives server  $Y$ 's local graph before server  $X$ 's. In this case, it would detect a cycle  $T \rightarrow U \rightarrow V \rightarrow T$ , although the edge  $T \rightarrow U$  no longer exists. This is an example of a phantom deadlock.

The observant reader will have realized that if transactions are using two-phase locks, they cannot release objects and then obtain more objects, and phantom deadlock cycles cannot occur in the way suggested above. Consider the situation in which a cycle

**Figure 13.14** Local and global wait-for graphs

$T \rightarrow U \rightarrow V \rightarrow T$  is detected: either this represents a deadlock or each of the transactions  $T$ ,  $U$  and  $V$  must eventually commit. It is actually impossible for any of them to commit, because each of them is waiting for an object that will never be released.

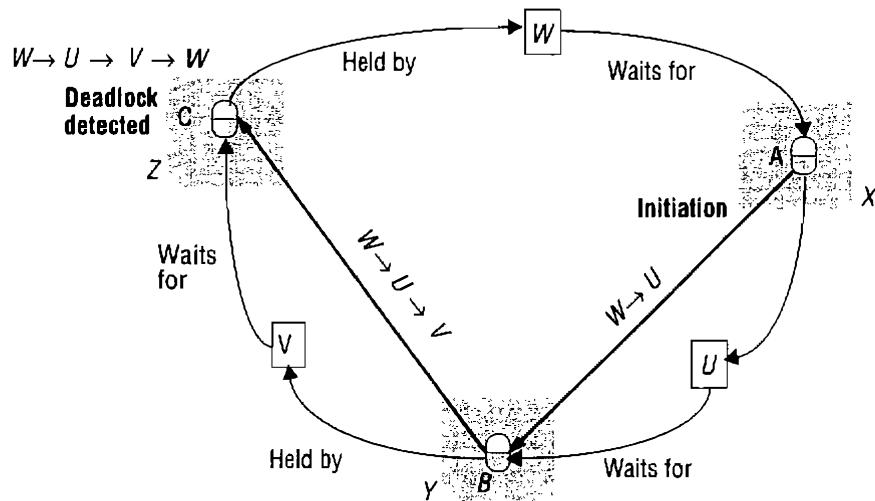
A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure. For example, if there is a cycle  $T \rightarrow U \rightarrow V \rightarrow T$  and  $U$  aborts after the information concerning  $U$  has been collected, then the cycle has been broken already and there is no deadlock.

**Edge chasing** ◊ A distributed approach to deadlock detection uses a technique called edge chasing or path pushing. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph.

The question is: when should a server send out a probe? Consider the situation at server  $X$  in Figure 13.13. This server has just added the edge  $W \rightarrow U$  to its local wait-for graph and at this time, transaction  $U$  is waiting to access object  $B$ , which transaction  $V$  holds at server  $Y$ . This edge could possibly be part of a cycle such as  $V \rightarrow T_1 \rightarrow T_2 \rightarrow \dots \rightarrow W \rightarrow U \rightarrow V$  involving transactions using objects at other servers. This indicates that there is a potential distributed deadlock cycle, which could be found by sending out a probe to server  $Y$ .

Now consider the situation a little earlier when server  $Z$  added the edge  $V \rightarrow W$  to its local graph: at this point in time,  $W$  is not waiting. Therefore, there would be no point in sending out a probe.

Each distributed transaction starts at a server (called the coordinator of the transaction) and moves to several other servers (called participants in the transaction), which can communicate with the coordinator. At any point in time, a transaction can be either active or waiting at just one of these servers. The coordinator is responsible for recording whether the transaction is active or is waiting for a particular object, and participants can get this information from their coordinator. Lock managers inform coordinators when transactions start waiting for objects and when transactions acquire objects and become active again. When a transaction is aborted to break a deadlock, its coordinator will inform the participants and all of its locks will be removed, with the effect that all edges involving that transaction will be removed from the local wait-for graphs.

**Figure 13.15** Probes transmitted to detect deadlock

Edge-chasing algorithms have three steps – initiation, detection and resolution.

*Initiation:* When a server notes that a transaction  $T$  starts waiting for another transaction  $U$ , where  $U$  is waiting to access an object at another server, it initiates detection by sending a probe containing the edge  $\langle T \rightarrow U \rangle$  to the server of the object at which transaction  $U$  is blocked. If  $U$  is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

*Detection:* Detection consists of receiving probes and deciding whether deadlock has occurred and whether to forward the probes.

For example, when a server of an object receives a probe  $\langle T \rightarrow U \rangle$  (indicating that  $T$  is waiting for a transaction  $U$  that holds a local object), it checks to see whether  $U$  is also waiting. If it is, the transaction it waits for (for example,  $V$ ) is added to the probe (making it  $\langle T \rightarrow U \rightarrow V \rangle$ ), and if the new transaction ( $V$ ) is waiting for another object elsewhere, the probe is forwarded.

In this way, paths through the global wait-for graph are built one edge at a time. Before forwarding a probe, the server checks to see whether the transaction (for example,  $T$ ) it has just added has caused the probe to contain a cycle (for example,  $\langle T \rightarrow U \rightarrow V \rightarrow T \rangle$ ). If this is the case, it has found a cycle in the graph and deadlock has been detected.

*Resolution:* When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

In our example, the following steps describe how deadlock detection is initiated and the probes that are forwarded during the corresponding detection phase.

- Server  $X$  initiates detection by sending probe  $\langle W \rightarrow U \rangle$  to the server of  $B$  (Server  $Y$ ).

- Server  $Y$  receives probe  $\langle W \rightarrow U \rangle$ , notes that  $B$  is held by  $V$  and appends  $V$  to the probe to produce  $\langle W \rightarrow U \rightarrow V \rangle$ . It notes that  $V$  is waiting for  $C$  at server  $Z$ . This probe is forwarded to server  $Z$ .
- Server  $Z$  receives probe  $\langle W \rightarrow U \rightarrow V \rangle$  and notes  $C$  is held by  $W$  and appends  $W$  to the probe to produce  $\langle W \rightarrow U \rightarrow V \rightarrow W \rangle$ .

This path contains a cycle. The server detects a deadlock. One of the transactions in the cycle must be aborted to break the deadlock. The transaction to be aborted can be chosen according to transaction priorities, which are described shortly.

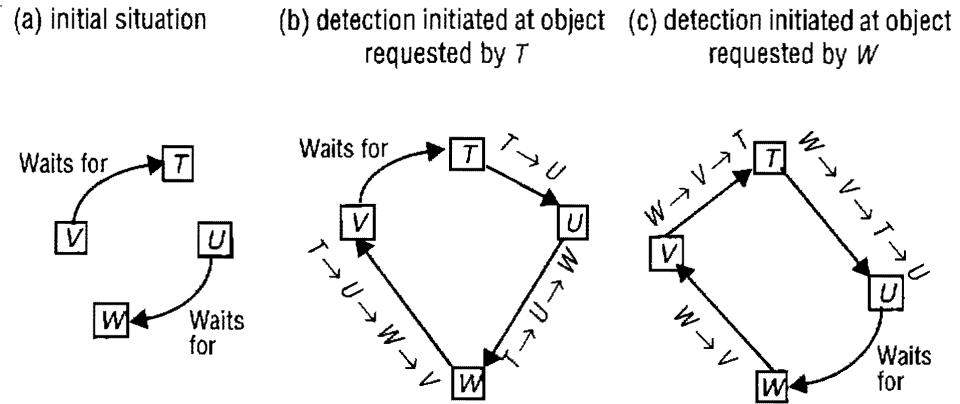
Figure 13.15 shows the progress of the probe messages from the initiation by the server of  $A$  to the deadlock detection by the server of  $C$ . Probes are shown as heavy arrows, objects as circles and transaction coordinators as rectangles. Each probe is shown as going directly from one object to another. In reality, before a server transmits a probe to another server, it consults the coordinator of the last transaction in the path to find out whether the latter is waiting for another object elsewhere. For example, before the server of  $B$  transmits the probe  $\langle W \rightarrow U \rightarrow V \rangle$  it consults the coordinator of  $V$  to find out that  $V$  is waiting for  $C$ . In most of the edge-chasing algorithms, the servers of objects send probes to transaction coordinators, which then forward them (if the transaction is waiting) to the server of the object the transaction is waiting for. In our example, the server of  $B$  transmits the probe  $\langle W \rightarrow U \rightarrow V \rangle$  to the coordinator of  $V$ , which then forwards it to the server of  $C$ . This shows that when a probe is forwarded, two messages are required.

The above algorithm should find any deadlock that occurs, provided that waiting transactions do not abort and there are no failures such as lost messages or servers crashing. To understand this, consider a deadlock cycle in which the last transaction,  $W$ , starts waiting and completes the cycle. When  $W$  starts waiting for an object, the server initiates a probe that goes to the server of the object held by each transaction that  $W$  is waiting for. The recipients extend and forward the probes to the servers of objects requested by all waiting transactions they find. Thus every transaction that  $W$  waits for directly or indirectly will be added to the probe unless a deadlock is detected. When there is a deadlock,  $W$  is waiting for itself indirectly. Therefore, the probe will return to the object that  $W$  holds.

It might appear that large numbers of messages are sent in order to detect deadlock. In the above example, we see two probe messages to detect a cycle involving three transactions. Each of the probe messages is in general two messages (from object to coordinator and then from coordinator to object).

A probe that detects a cycle involving  $N$  transactions will be forwarded by  $(N - 1)$  transaction coordinators via  $(N - 1)$  servers of objects, requiring  $2(N - 1)$  messages. Fortunately, the majority of deadlocks involve cycles containing only two transactions, and there is no need for undue concern about the number of messages involved. This observation has been made from studies of databases. It can also be argued by considering the probability of conflicting access to objects. See Bernstein *et al.* [1987].

**Transaction priorities** ♦ In the above algorithm, every transaction involved in a deadlock cycle can cause deadlock detection to be initiated. The effect of several transactions in a cycle initiating deadlock detection is that detection may happen at several different servers in the cycle with the result that more than one transaction in the cycle is aborted.

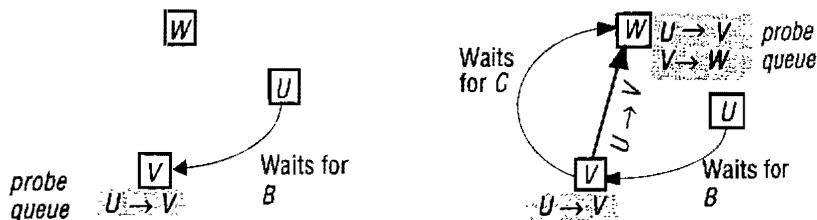
**Figure 13.16** Two probes initiated

In Figure 13.16(a), consider transactions  $T, U, V$  and  $W$ , where  $U$  is waiting for  $W$  and  $V$  is waiting for  $T$ . At about the same time,  $T$  requests the object held by  $U$  and  $W$  requests the object held by  $V$ . Two separate probes  $\langle T \rightarrow U \rangle$  and  $\langle W \rightarrow V \rangle$  are initiated by the servers of these objects and are circulated until deadlock is detected by each of two different servers. See in Figure 13.16(b), where the cycle is  $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$ , and (c), where the cycle is  $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$ .

In order to ensure that only one transaction in a cycle is aborted, transactions are given *priorities* in such a way that all transactions are totally ordered. Timestamps for example, may be used as priorities. When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even if several different servers detect the same cycle, they will all reach the same decision as to which transaction is to be aborted. We write  $T > U$  to indicate that  $T$  has higher priority than  $U$ . In the above example, assume  $T > U > V > W$ . Then the transaction  $W$  will be aborted when either of the cycles  $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$  or  $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$  is detected.

It might appear that transaction priorities could also be used to reduce the number of situations that cause deadlock detection to be initiated, by using the rule that detection is initiated only when a higher-priority transaction starts to wait for a lower-priority one. In our example in Figure 13.16, as  $T > U$  the initiating probe  $\langle T \rightarrow U \rangle$  would be sent, but as  $W < V$  the initiating probe  $\langle W \rightarrow V \rangle$  would not be sent. If we assume that when a transaction starts waiting for another transaction it is equally likely that the waiting transaction has higher or lower priority than the waited-for transaction, then the use of this rule is likely to reduce the number of probe messages by about half.

Transaction priorities could also be used to reduce the number of probes that are forwarded. The general idea is that probes should travel ‘downhill’ – that is, from transactions with higher priorities to transactions with lower priorities. To do this, servers use the rule that they do not forward any probe to a holder that has higher priority than the initiator. The argument for doing this is that if the holder is waiting for another transaction then it must have initiated detection by sending a probe when it started waiting.

**Figure 13.17** Probes travel downhill(a)  $V$  stores probe when  $U$  starts waiting (b) Probe is forwarded when  $V$  starts waiting

However, there is a pitfall associated with these apparent improvements. In our example in Figure 13.15 transactions  $U$ ,  $V$  and  $W$  are executed in an order in which  $U$  is waiting for  $V$  and  $V$  is waiting for  $W$  when  $W$  starts waiting for  $U$ . Without priority rules, detection is initiated when  $W$  starts waiting by sending a probe  $<W \rightarrow U>$ . Under the priority rule, this probe will not be sent, because  $W < U$  and deadlock will not be detected.

The problem is that the order in which transactions start waiting can determine whether or not deadlock will be detected. The above pitfall can be avoided by using a scheme in which coordinators save copies of all the probes received on behalf of each transaction in a *probe queue*. When a transaction starts waiting for an object, it forwards the probes in its queue to the server of the object, which propagates the probes on downhill routes.

In our example in Figure 13.15, when  $U$  starts waiting for  $V$ , the coordinator of  $V$  will save the probe  $<U \rightarrow V>$ . See Figure 13.17(a). Then when  $V$  starts waiting for  $W$ , the coordinator of  $W$  will store  $<V \rightarrow W>$  and  $V$  will forward its probe queue  $<U \rightarrow V>$  to  $W$ . See Figure 13.17(b), in which  $W$ 's probe queue has  $<U \rightarrow V>$  and  $<V \rightarrow W>$ . When  $W$  starts waiting for  $A$  it will forward its probe queue  $<U \rightarrow V \rightarrow W>$  to the server of  $A$ , which also notes the new dependency  $W \rightarrow U$  and combines it with the information in the probe received to determine that  $U \rightarrow V \rightarrow W \rightarrow U$ . Deadlock is detected.

When an algorithm requires probes to be stored in probe queues, it also requires arrangements to pass on probes to new holders and to discard probes that refer to transactions that have been committed or aborted. If relevant probes are discarded, undetected deadlocks may occur, and if outdated probes are retained, false deadlocks may be detected. This adds much to the complexity of any edge-chasing algorithm. Readers who are interested in the details of such algorithms should see Sinha and Natarajan [1985] and Choudhary *et al.* [1989], who present algorithms for use with exclusive locks. But they will see that Choudhary *et al.* showed that Sinha and Natarajan's algorithm is incorrect and fails to detect all deadlocks and may even report false deadlocks. Kshemkalyani and Singhal [1991] corrected the algorithm of Choudhary *et al.* (which fails to detect all deadlocks and may report false deadlocks) and provide a proof of correctness for the corrected algorithm. In a subsequent paper,

Kshemkalyani and Singhal [1994] argue that distributed deadlocks are not very well understood because there is no global state or time in a distributed system. In fact, any cycle that has been collected may contain sections recorded at different times. In addition, sites may hear about deadlocks but may not hear that they have been resolved until after random delays. The paper describes distributed deadlocks in terms of the contents of distributed memory, using causal relationships between events at different sites.

## 13.6 Transaction recovery

The atomic property of transactions requires that the effects of all committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects: durability and failure atomicity. Durability requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore, an acknowledgment of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects. Failure atomicity requires that effects of transactions are atomic even when the server crashes. Recovery is concerned with ensuring that a server's objects are durable and that the service provides failure atomicity.

Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes. In this chapter, we assume that when a server is running it keeps all of its objects in its volatile memory and records its committed objects in a *recovery file* or files. Therefore, recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.

The two requirements for durability and for failure atomicity are not really independent of one another and can be dealt with by a single mechanism – the *recovery manager*. The task of a recovery manager is:

- to save objects in permanent storage (in a recovery file) for committed transactions;
- to restore the server's objects after a crash;
- to reorganize the recovery file to improve the performance of recovery;
- to reclaim storage space (in the recovery file).

In some cases, we require the recovery manager to be resilient to media failures – failures of its recovery file so that some of the data on the disk is lost, either by being corrupted during a crash, by random decay or by a permanent failure. In such cases, we need another copy of the recovery file. This can be in stable storage, which is implemented so as to be very unlikely to fail by using mirrored disks or copies at a different location.

**Intentions list** ◊ Any server that provides transactions needs to keep track of the objects accessed by clients' transactions. Recall from Chapter 12 that when a client opens a transaction, the server first contacted provides a new transaction identifier and returns it to the client. Each subsequent client request within a transaction up to and including the *commit* or *abort* request includes the transaction identifier as an argument. During the progress of a transaction, the update operations are applied to a private set of tentative versions of the objects belonging to the transaction.

At each server, an *intentions list* is recorded for all of its currently active transactions – an intentions list of a particular transaction contains a list of the references and the values of all the objects that are altered by that transaction. When a transaction is committed, that transaction's intentions list is used to identify the objects it affected. The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's recovery file. When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction.

Recall also that a distributed transaction must carry out an atomic commit protocol before it can be committed or aborted. Our discussion of recovery is based on the two-phase commit protocol, in which all the participants involved in a transaction first say whether they are prepared to commit and then, later on if all the participants agree, they all carry out the actual commit actions. If the participants cannot agree to commit, they must abort the transaction.

At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file, so that it will be able to carry out the commitment later on, even if it crashes in the interim.

When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction. Once the client has been informed that a transaction has committed, the recovery files of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

**Entries in recovery file** ◊ To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the values of the objects is stored in the recovery file. This information concerns the *status* of each transaction – whether it is *committed*, *aborted* or *prepared* to commit. In addition, each object in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file. Figure 13.18 shows a summary of the types of entry included in a recovery file.

The transaction status values relating to the two-phase commit protocol are discussed in Section 13.6.4 on recovery of the two-phase commit protocol. We shall now describe two approaches to the use of recovery files: logging and shadow versions.

### 13.6.1 Logging

In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server. The history consists of values of objects,

**Figure 13.18** Types of entry in a recovery file

Type of entry	Description of contents of entry
Object	A value of an object.
Transaction status	Transaction identifier, transaction status ( <i>prepared</i> , <i>committed</i> , <i>aborted</i> ) – and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, <position in recovery file of value of object>.

transaction status entries and intentions lists of transactions. The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server. In practice, the recovery file will contain a recent snapshot of the values of all the objects in the server followed by a history of transactions after the snapshot.

During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (*prepared*) together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file.

It is assumed that the append operation is atomic in the sense that it writes one or more complete entries to the recovery file. If the server fails, only the last write can be incomplete. To make efficient use of the disk, several subsequent writes can be buffered and then written as a single write to disk. An additional advantage of the logging technique is that sequential writes to disk are faster than writes to random locations.

After a crash, any transaction that does not have a *committed* status in the log is aborted. Therefore, when a transaction commits, its *committed* status entry must be *forced* to the log – that is, written to the log together with any other buffered entries.

The recovery manager associates a unique identifier with each object so that the successive versions of an object in the recovery file may be associated with the server's objects. For example, a durable form of a remote object reference such as a CORBA persistent reference will do as an object identifier.

Figure 13.19 illustrates the log mechanism for the banking service transactions *T* and *U* in Figure 12.7. The log was recently reorganized, and entries to the left of the double line represent a snapshot of the values of *A*, *B* and *C* before transactions *T* and *U* started. In this diagram, we use the names *A*, *B* and *C* as unique identifiers for objects. We show the situation when transaction *T* has committed and transaction *U* has prepared but not committed. When transaction *T* prepares to commit, the values of objects *A* and *B* are written at positions *P*<sub>1</sub> and *P*<sub>2</sub> in the log, followed by a prepared transaction status entry for *T* with its intentions list (<*A*, *P*<sub>1</sub>>, <*B*, *P*<sub>2</sub>>). When transaction *T* commits, a committed transaction status entry for *T* is put at position *P*<sub>4</sub>. Then when transaction *U*

**Figure 13.19** Log for banking service

$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
Object: <i>A</i> 100	Object: <i>B</i> 200	Object: <i>C</i> 300	Object: <i>A</i> 80	Object: <i>B</i> 220	Trans: <i>T</i> prepared $\langle A, P_1 \rangle$ $\langle B, P_2 \rangle$ $P_0$	Trans: <i>T</i> committed $P_3$	Object: <i>C</i> 278
						Object: <i>B</i> 242	Trans: <i>U</i> prepared $\langle C, P_5 \rangle$ $\langle B, P_6 \rangle$ $P_4$

The diagram shows the log structure with checkpoints and end of log markers. Three curved arrows point from the right side of the log to specific points: one arrow points to the boundary between  $P_3$  and  $P_4$  labeled 'Checkpoint'; another arrow points to the boundary between  $P_6$  and  $P_7$  labeled 'End of log'.

prepares to commit, the values of objects *C* and *B* are written at positions  $P_5$  and  $P_6$  in the log, followed by a prepared transaction status entry for *U* with its intentions list ( $\langle C, P_5 \rangle, \langle B, P_6 \rangle$ ).

Each transaction status entry contains a pointer to the position in the recovery file of the previous transaction status entry to enable the recovery manager to follow the transaction status entries in reverse order through the recovery file. The last pointer in the sequence of transaction status entries points to the checkpoint.

**Recovery of objects** ◊ When a server is replaced after a crash, it first sets default initial values for its objects and then hands over to its recovery manager. The recovery manager is responsible for restoring the server's objects so that they include all the effects of all the committed transactions performed in the correct order and none of the effects of incomplete or aborted transactions.

The most recent information about transactions is at the end of the log. There are two approaches to restoring the data from the recovery file. In the first, the recovery manager starts at the beginning and restores the values of all of the objects from the most recent checkpoint. It then reads in the values of each of the objects, associates them with their intentions lists and for committed transactions replaces the values of the objects. In this approach, the transactions are replayed in the order in which they were executed and there could be a large number of them. In the second approach, the recovery manager will restore a server's objects by 'reading the recovery file backwards'. The recovery file has been structured so that there is a backwards pointer from each transaction status entry to the next. The recovery manager uses transactions with committed status to restore those objects that have not yet been restored. It continues until it has restored all of the server's objects. This has the advantage that each object is restored once only.

To recover the effects of a transaction, a recovery manager gets the corresponding intentions list from its recovery file. The intentions list contains the identifiers and positions in the recovery file of values of all the objects affected by the transaction.

If the server fails at the point reached in Figure 13.19, its recovery manager will recover the objects as follows. It starts at the last transaction status entry in the log (at  $P_7$ ) and concludes that transaction *U* has not committed and its effects should be ignored. It then moves to the previous transaction status entry in the log (at  $P_4$ ) and concludes that transaction *T* has committed. To recover the objects affected by

transaction  $T$ , it moves to the previous transaction status entry in the log (at  $P_3$ ) and finds the intentions list for  $T (< A, P_1 >, < B, P_2 >)$ . It then restores objects  $A$  and  $B$  from the values at  $P_1$  and  $P_2$ . As it has not yet restored  $C$ , it moves back to  $P_0$ , which is a checkpoint, and restores  $C$ .

To help with subsequent reorganization of the recovery file, the recovery manager notes all the prepared transactions it finds during the process of restoring the server's objects. For each prepared transaction, it adds an aborted transaction status to the recovery file. This ensures that in the recovery file, every transaction is eventually shown as either committed or aborted.

The server could fail again during the recovery procedures. It is essential that recovery be idempotent in the sense that it can be done any number of times with the same effect. This is straightforward under our assumption that all the objects are restored to volatile memory. In the case of a database, which keeps its objects in permanent storage, with a cache in volatile memory, some of the objects in permanent storage will be out of date when a server is replaced after a crash. Therefore, its recovery manager has to restore the objects in permanent storage. If it fails during recovery, the partially restored objects will still be there. This makes idempotence a little harder to achieve.

**Reorganizing the recovery file** ◊ A recovery manager is responsible for reorganizing its recovery file so as to make the process of recovery faster and to reduce its use of space. If the recovery file is never reorganized, then the recovery process must search backwards through the recovery file until it has found a value for each of its objects. Conceptually, the only information required for recovery is a copy of the committed versions of all the objects in the server. This would be the most compact form for the recovery file. The name *checkpointing* is used to refer to the process of writing the current committed values of a server's objects to a new recovery file, together with transaction status entries and intentions lists of transactions that have not yet been fully resolved (including information related to the two-phase commit protocol). The term *checkpoint* is used to refer to the information stored by the checkpointing process. The purpose of making checkpoints is to reduce the number of transactions to be dealt with during recovery and to reclaim file space.

Checkpointing can be done immediately after recovery but before any new transactions are started. However, recovery may not occur very often. Therefore, checkpointing may need to be done from time to time during the normal activity of a server. The checkpoint is written to a future recovery file, and the current recovery file remains in use until the checkpoint is complete. Checkpointing consists of 'adding a mark' to the recovery file when the checkpointing starts, writing the server's objects to the future recovery file and then copying (1) entries before the mark that relate to as yet unresolved transactions and (2) all entries after the mark in the recovery file to the future recovery file. When the checkpoint is complete, the future recovery file becomes the recovery file.

The recovery system can reduce its use of space by discarding the old recovery file. When the recovery manager is carrying out the recovery process, it may encounter a checkpoint in the recovery file. When this happens, it can restore immediately all outstanding objects from the checkpoint.

**Figure 13.20** Shadow versions

	Map at start			Map when T commits			
	$P_0$	$P_0'$	$P_0''$	$P_1$	$P_2$	$P_3$	$P_4$
<i>Version store</i>	100	200	300	80	220	278	242
	<i>Checkpoint</i>						

### 13.6.2 Shadow versions

The *logging* technique records transaction status entries, intentions lists and objects all in the same file – the log. The *shadow versions* technique is an alternative way to organize a recovery file. It uses a *map* to locate versions of the server's objects in a file called a *version store*. The map associates the identifiers of the server's objects with the positions of their current versions in the version store. The versions written by each transaction are shadows of the previous committed versions. The transaction status entries and intentions lists are dealt with separately. Shadow versions are described first.

When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged. These new as yet tentative versions are called *shadow* versions. When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. To complete the commit process, the new map replaces the old map.

To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store.

This technique is illustrated with the same example involving transactions *T* and *U* in Figure 13.20. The first column in the table shows the map before transactions *T* and *U*, when the balances of the accounts *A*, *B* and *C* are \$100, \$200 and \$300, respectively. The second column shows the map after transaction *T* has committed.

The version store contains a checkpoint, followed by the versions of *A* and *B* at  $P_1$  and  $P_2$  made by transaction *T*. It also contains the shadow versions of *B* and *C* made by transaction *U*, at  $P_3$  and  $P_4$ .

The map must always be written to a well-known place (for example, at the start of the version store or a separate file) so that it can be found when the system needs to be recovered.

The switch from the old map to the new map must be performed in a single atomic step. To achieve this it is essential that stable storage is used for the map – so that there is guaranteed to be a valid map even when a file write operation fails. The shadow versions method provides faster recovery than logging because the positions of the current committed objects are recorded in the map, whereas recovery from a log requires searching throughout the log for objects. Logging should be faster than shadow versions

during the normal activity of the system. This is because logging requires only a sequence of append operations to the same file, whereas shadow versions requires an additional stable storage write (involving two unrelated disk blocks).

Shadow versions on their own are not sufficient for a server that handles distributed transactions. Transaction status entries and intentions lists are saved in a file called the transaction status file. Each intentions list represents the part of the map that will be altered by a transaction when it commits. The transaction status file may, for example, be organized as a log.

The figure below shows the map and the transaction status file for our current example when  $T$  has committed and  $U$  is prepared to commit.

<i>Map</i>	<i>Stable storage</i>	<i>T</i>	<i>T</i>	<i>U</i>
$A \rightarrow P_1$		prepared	committed	prepared
$B \rightarrow P_2$		$A \rightarrow P_1$		$B \rightarrow P_3$
$C \rightarrow P_0$	<i>Transaction status file</i>	$B \rightarrow P_2$		$C \rightarrow P_4$

There is a chance that a server may crash between the time when a committed status is written to the transaction status file and the time when the map is updated – in which case the client will not have been acknowledged. The recovery manager must allow for this possibility when the server is replaced after a crash, for example by checking whether the map includes the effects of the last committed transaction in the transaction status file. If it does not, then the latter should be marked as aborted.

### 13.6.3 The need for transaction status and intentions list entries in a recovery file

It is possible to design a simple recovery file that does not include entries for transaction status items and intentions lists. This sort of recovery file may be suitable when all transactions are directed to a single server. The use of transaction status items and intentions lists in the recovery file is essential for a server that is intended to participate in distributed transactions. This approach can also be useful for servers of non-distributed transactions for various reasons, including the following:

- Some recovery managers are designed to write the objects to the recovery file early – under the assumption that transactions normally commit.
- If transactions use a large number of big objects, the need to write them contiguously to the recovery file may complicate the design of a server. When objects are referenced from intentions lists, they can be found wherever they are.
- In timestamp ordering concurrency control, a server sometimes knows that a transaction will eventually be able to commit and acknowledges the client – at this time the objects are written to the recovery file (see Chapter 12) to ensure their permanence. However, the transaction may have to wait to commit until earlier transactions have committed. In such situations, the corresponding transaction status entries in the recovery file will be *waiting to commit* and then *committed* to ensure timestamp ordering of committed transactions in the recovery file. On recovery, any waiting-to-commit transactions can be allowed to commit, because

**Figure 13.21** Log with entries relating to two-phase commit protocol

Trans: $T$ prepared intentions list	Coord'r: $T$ part'part list: . . .	•	Trans: $T$ committed	Trans: $U$ prepared intentions list	•	Part'part: $U$ Coord'r: . . .	Trans: $U$ uncertain	Trans: $U$ committed
--	--	---	-------------------------	--	---	----------------------------------	-------------------------	-------------------------

the ones they were waiting for have either just committed or if not have to be aborted due to failure of the server.

### 13.6.4 Recovery of the two-phase commit protocol

In a distributed transaction, each server keeps its own recovery file. The recovery management described in the previous section must be extended to deal with any transactions that are performing the two-phase commit protocol at the time when a server fails. The recovery managers use two new status values: *done*, *uncertain*. These status values are shown in Figure 13.6. A coordinator uses *committed* to indicate that the outcome of the vote is *Yes* and *done* to indicate that the two-phase commit protocol is complete. A participant uses *uncertain* to indicate that it has voted *Yes* but does not yet know the outcome. Two additional types of entry allow a coordinator to record a list of participants and a participant to record its coordinator:

Type of entry	Description of contents of entry
<i>Coordinator</i>	Transaction identifier, list of participants
<i>Participant</i>	Transaction identifier, coordinator

In phase 1 of the protocol, when the coordinator is prepared to commit (and has already added a prepared status entry to its recovery file), its recovery manager adds a *coordinator* entry to its recovery file. Before a participant can vote *Yes*, it must have already prepared to commit (and must have already added a prepared status entry to its recovery file). When it votes *Yes*, its recovery manager records a *participant* entry and adds an *uncertain* transaction status to its recovery file as a forced write. When a participant votes *No*, it adds an *abort* transaction status to its recovery file.

In phase 2 of the protocol, the recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision. This must be a forced write. Recovery managers of participants add a *commit* or *abort* transaction status to their recovery files according to the message received from the coordinator. When a coordinator has received a confirmation from all of its participants, its recovery manager adds a *done* transaction status to its recovery file - this need not be forced. The *done* status entry is not part of the protocol but is used when the recovery file is reorganized. Figure 13.21 shows the entries in a log for transaction  $T$ , in which the server played the coordinator role, and for transaction  $U$ , in which the server played the

**Figure 13.22** Recovery of the two-phase commit protocol

<i>Role</i>	<i>Status</i>	<i>Action of recovery manager</i>
Coordinator	<i>prepared</i>	No decision had been reached before the server failed. It sends <i>abortTransaction</i> to all the servers in the participant list and adds the transaction status <i>aborted</i> in its recovery file. Same action for state <i>aborted</i> . If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	<i>committed</i>	A decision to commit had been reached before the server failed. It sends a <i>doCommit</i> to all of the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (see Figure 13.5).
Participant	<i>committed</i>	The participant sends a <i>haveCommitted</i> message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	<i>uncertain</i>	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a <i>getDecision</i> to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	<i>prepared</i>	The participant has not yet voted and can abort the transaction.
Coordinator	<i>done</i>	No action is required.

participant role. For both transactions, the *prepared* transaction status entry comes first. In the case of a coordinator it is followed by a coordinator entry, and a *committed* transaction status entry. The *done* transaction status entry is not shown in Figure 13.21. In the case of a participant, the *prepared* transaction status entry is followed by a participant entry whose state is *uncertain* and then a *committed* or *aborted* transaction status entry.

When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restoring the objects. For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server played the participant role, it should find a participant entry and a set of transaction status entries. In both cases, the most recent transaction status entry – that is, the one nearest the end of the log – determines the transaction status at the time of failure. The action of the recovery manager with respect to the two-phase commit protocol for any transaction

depends on whether the server was the coordinator or a participant and on its status at the time of failure, as shown in Figure 13.22.

**Reorganization of recovery file** ◊ Care must be taken when performing a checkpoint to ensure that *coordinator* entries of transactions without status *done* are not removed from the recovery file. These entries must be retained until all the participants have confirmed that they have completed their transactions. Entries with status *done* may be discarded. Participant entries with transaction state *uncertain* must also be retained.

**Recovery of nested transactions** ◊ In the simplest case, each subtransaction of a nested transaction accesses a different set of objects. As each participant prepares to commit during the two-phase commit protocol, it writes its objects and intentions lists to the local recovery file, associating them with the transaction identifier of the top-level transaction. Although nested transactions use a special variant of the two-phase commit protocol, the recovery manager uses the same transaction status values as for flat transactions.

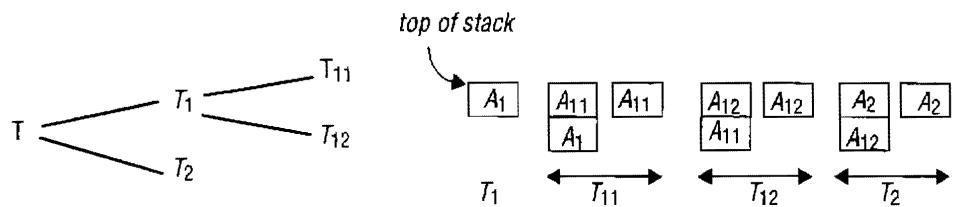
However, abort recovery is complicated by the fact that several subtransactions at the same and different levels in the nesting hierarchy can access the same object. Section 12.4 describes a locking scheme in which parent transactions inherit locks and subtransactions acquire locks from their parents. The locking scheme forces parent transactions and subtransactions to access common data objects at different times and ensures that accesses by concurrent subtransactions to the same objects must be serialized.

Objects that are accessed according to the rules of nested transactions are made recoverable by providing tentative versions for each subtransaction. The relationship between the tentative versions of an object used by the subtransactions of a nested transaction is similar to the relationship between the locks. To support recovery from aborts, the server of an object shared by transactions at multiple levels provides a stack of tentative versions – one for each nested transaction to use.

When the first subtransaction in a set of nested transactions accesses an object, it is provided with a tentative version that is a copy of the current committed version of the object. This is regarded as being at the top of the stack, but unless any other subtransactions access the same object, the stack will not materialize.

When one of its subtransactions does access the same object, it copies the version at the top of the stack and pushes it back on the stack. All of that subtransaction's updates are applied to the tentative version at the top of the stack. When a subtransaction provisionally commits, its parent inherits the new version. To achieve this, both the subtransaction's version and its parent's version are discarded from the stack and then the subtransaction's new version is pushed back on to the stack (effectively replacing its parent's version). When a subtransaction aborts, its version at the top of the stack is discarded. Eventually, when the top-level transaction commits, the version at the top of the stack (if any) becomes the new committed version.

For example, in Figure 13.23, suppose that transactions  $T_1$ ,  $T_{11}$ ,  $T_{12}$  and  $T_2$  all access the same object,  $A$ , in the order  $T_1$ ;  $T_{11}$ ;  $T_{12}$ ;  $T_2$ . Suppose that their tentative versions are called  $A_1$ ,  $A_{11}$ ,  $A_{12}$  and  $A_2$ . When  $T_1$  starts executing,  $A_1$  is based on the committed version of  $A$  and is pushed on the stack. When  $T_{11}$  starts executing, it bases its version  $A_{11}$  on  $A_1$  and pushes it on the stack; when it completes, it replaces its parent's

**Figure 13.23** Nested transactions

version on the stack. Transactions  $T_{12}$  and  $T_2$  act in a similar way, finally leaving the result of  $T_2$  at the top of the stack.

## 13.7 Summary

In the most general case, a client's transaction will request operations on objects in several different servers. A distributed transaction is any transaction whose activity involves several different servers. A nested transaction structure may be used to allow additional concurrency and independent committing by the servers in a distributed transaction.

The atomicity property of transactions requires that the servers participating in a distributed transaction either all commit it or all abort it. Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution. The two-phase commit protocol allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to servers crashing. The two-phase commit protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.

Concurrency control in distributed transactions is modular – each server is responsible for the serializability of transactions that access its own objects. However, additional protocols are required to ensure that transactions are serializable globally. Distributed transactions that use timestamp ordering require a means of generating an agreed timestamp ordering between the multiple servers. Those that use optimistic concurrency control require global validation or a means of forcing a global ordering on committing transactions.

Distributed transactions that use two-phase locking can suffer from distributed deadlocks. The aim of distributed deadlock detection is to look for cycles in the global wait-for graph. If a cycle is found, one or more transactions must be aborted to resolve the deadlock. Edge chasing is a non-centralized approach to the detection of distributed deadlocks.

Transaction-based applications have strong requirements for the long life and integrity of the information stored, but they do not usually have requirements for immediate response at all times. Atomic commit protocols are the key to distributed transactions, but they cannot be guaranteed to complete within a particular time limit. Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash. Users of a transaction service would experience some delay during recovery. Although it is

assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system, they are able to reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers.

## EXERCISES

- 13.1 In a decentralized variant of the two-phase commit protocol the participants communicate directly with one another instead of indirectly via the coordinator. In phase 1, the coordinator sends its vote to all the participants. In phase 2, if the coordinator's vote is *No*, the participants just abort the transaction; if it is *Yes*, each participant sends its vote to the coordinator and the other participants, each of which decides on the outcome according to the vote and carries it out. Calculate the number of messages and the number of rounds it takes. What are its advantages or disadvantages in comparison with the centralized variant? *page 520*
- 13.2 A three-phase commit protocol has the following parts:
- Phase 1:* is the same as for two-phase commit.
- Phase 2:* the coordinator collects the votes and makes a decision; if it is *No*, it *aborts* and informs participants that voted *Yes*; if the decision is *Yes*, it sends a *preCommit* request to all the participants. Participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests and carry out *doAbort* requests.
- Phase 3:* the coordinator collects the acknowledgments. When all are received, it *Commits* and sends *doCommit* to the participants. Participants wait for a *doCommit* request. When it arrives they *Commit*.
- Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants. Assume that communication does not fail. *page 523*
- 13.3 Explain how the two-phase commit protocol for nested transactions ensures that if the top-level transaction commits, all the right descendants are committed or aborted. *page 524*
- 13.4 Give an example of the interleavings of two transactions that is serially equivalent at each server but is not serially equivalent globally. *page 528*
- 13.5 The *getDecision* procedure defined in Figure 13.4 is provided only by coordinators. Define a new version of *getDecision* to be provided by participants for use by other participants that need to obtain a decision when the coordinator is unavailable.
- Assume that any active participant can make a *getDecision* request to any other active participant. Does this solve the problem of delay during the 'uncertain' period? Explain your answer. At what point in the two-phase commit protocol would the coordinator inform the participants of the other participants' identities (to enable this communication)? *page 520*

- 
- 13.6 Extend the definition of two-phase locking to apply to distributed transactions. Explain how this is ensured by distributed transactions using strict two-phase locking locally.  
*page 528 and Chapter 12*
- 13.7 Assuming that strict two-phase locking is in use, describe how the actions of the two-phase commit protocol relate to the concurrency control actions of each individual server. How does distributed deadlock detection fit in?  
*pages 520 and 528*
- 13.8 A server uses timestamp ordering for local concurrency control. What changes must be made to adapt it for use with distributed transactions? Under what conditions could it be argued that the two-phase commit protocol is redundant with timestamp ordering?  
*pages 520 and 529*
- 13.9 Consider distributed optimistic concurrency control in which each server performs local backward validation sequentially (that is, with only one transaction in the validate and update phase at one time), in relation to your answer to Exercise 13.4. Describe the possible outcomes when the two transactions attempt to commit. What difference does it make if the servers use parallel validation?  
*Chapter 12 and page 530*
- 13.10 A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure.  
*page 533*
- 13.11 Consider the edge-chasing algorithm (without priorities). Give examples to show that it could detect phantom deadlocks.  
*page 534*
- 13.12 A server manages the objects  $a_1, a_2, \dots, a_n$ . It provides two operations for its clients:  
 $Read(i)$  returns the value of  $a_i$   
 $Write(i, Value)$  assigns  $Value$  to  $a_i$   
The transactions  $T$ ,  $U$  and  $V$  are defined as follows:  
 $T: x = Read(i); Write(j, 44);$   
 $U: Write(i, 55); Write(j, 66);$   
 $V: Write(k, 77); Write(k, 88);$   
Describe the information written to the log file on behalf of these three transactions if strict two-phase locking is in use and  $U$  acquires  $a_i$  and  $a_j$  before  $T$ . Describe how the recovery manager would use this information to recover the effects of  $T$ ,  $U$  and  $V$  when the server is replaced after a crash. What is the significance of the order of the commit entries in the log file?  
*pages 540-542*
- 13.13 The appending of an entry to the log file is atomic, but append operations from different transactions may be interleaved. How does this affect the answer to Exercise 13.12?  
*pages 540-542*

- 13.14 The transactions  $T$ ,  $U$  and  $V$  of Exercise 13.12 use strict two-phase locking and their requests are interleaved as follows:

$T$	$U$	$V$
$x = \text{Read}(i);$		
	$\text{Write}(i, 55)$	$\text{Write}(k, 77);$
$\text{Write}(j, 44)$		$\text{Write}(k, 88)$
	$\text{Write}(j, 66)$	

Assuming that the recovery manager appends the data entry corresponding to each  $\text{Write}$  operation to the log file immediately instead of waiting until the end of the transaction, describe the information written to the log file on behalf of the transactions  $T$ ,  $U$  and  $V$ . Does early writing affect the correctness of the recovery procedure? What are the advantages and disadvantages of early writing?

pages 540–542

- 13.15 Transactions  $T$  and  $U$  are run with timestamp ordering concurrency control. Describe the information written to the log file on behalf of  $T$  and  $U$ , allowing for the fact that  $U$  has a later timestamp than  $T$  and must wait to commit after  $T$ . Why is it essential that the commit entries in the log file be ordered by timestamps? Describe the effect of recovery if the server crashes (i) between the two *Commits* and (ii) after both of them.

$T$	$U$
$x = \text{Read}(i);$	
	$\text{Write}(i, 55);$
	$\text{Write}(j, 66);$
$\text{Write}(j, 44);$	
	<i>Commit</i>
	<i>Commit</i>

What are the advantages and disadvantages of early writing with timestamp ordering?

page 545

- 13.16 The transactions  $T$  and  $U$  in Exercise 13.15 are run with optimistic concurrency control using backward validation and restarting any transactions that fail. Describe the information written to the log file on their behalf. Why is it essential that the commit entries in the log file be ordered by transaction numbers? How are the write sets of committed transactions represented in the log file?

pages 540–542

- 13.17 Suppose that the coordinator of a transaction crashes after it has recorded the intentions list entry but before it has recorded the participant list or sent out the *canCommit?* requests. Describe how the participants resolve the situation. What will the coordinator do when it recovers? Would it be any better to record the participant list before the intentions list entry?

page 546

# Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial

FRED B. SCHNEIDER

*Department of Computer Science, Cornell University, Ithaca, New York 14853*

The state machine approach is a general method for implementing fault-tolerant services in distributed systems. This paper reviews the approach and describes protocols for two different failure models—Byzantine and fail stop. System reconfiguration techniques for removing faulty components and integrating repaired components are also discussed.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—*network operating systems*; D.2.10 [Software Engineering]: Design—*methodologies*; D.4.5 [Operating Systems]: Reliability—*fault tolerance*; D.4.7 [Operating Systems]: Organization and Design—*interactive systems, real-time systems*

General Terms: Algorithms, Design, Reliability

Additional Key Words and Phrases: Client-server, distributed services, state machine approach

## INTRODUCTION

Distributed software is often structured in terms of *clients* and *services*. Each service comprises one or more *servers* and exports *operations* that clients invoke by making *requests*. Although using a single, centralized, server is the simplest way to implement a service, the resulting service can only be as fault tolerant as the processor executing that server. If this level of fault tolerance is unacceptable, then multiple servers that fail independently must be used. Usually, replicas of a single server are executed on separate processors of a distributed system, and protocols are used to coordinate client interactions with these replicas. The physical and electrical isolation of processors in a distributed system ensures that server failures are independent, as required.

The *state machine approach* is a general method for implementing a fault-tolerant

service by replicating servers and coordinating client interactions with server replicas.<sup>1</sup> The approach also provides a framework for understanding and designing replication management protocols. Many protocols that involve replication of data or software—be it for masking failures or simply to facilitate cooperation without centralized control—can be derived using the state machine approach. Although few of these protocols actually were obtained in this manner, viewing them in terms of state machines helps in understanding how and why they work.

This paper is a tutorial on the state machine approach. It describes the approach and its implementation for two representative environments. Small examples suffice to illustrate the points. However, the

<sup>1</sup> The term “state machine” is a poor one, but, nevertheless, is the one used in the literature.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.  
© 1990 ACM 0360-0300/90/1200-0299 \$01.50

## CONTENTS

INTRODUCTION
1. STATE MACHINES
2. FAULT TOLERANCE
3. FAULT-TOLERANT STATE MACHINES
3.1 Agreement
3.2 Order and Stability
4. TOLERATING FAULTY OUTPUT DEVICES
4.1 Outputs Used Outside the System
4.2 Outputs Used Inside the System
5. TOLERATING FAULTY CLIENTS
5.1 Replicating the Client
5.2 Defensive Programming
6. USING TIME TO MAKE REQUESTS
7. RECONFIGURATION
7.1 Managing the Configuration
7.2 Integrating a Repaired Object
8. RELATED WORK
ACKNOWLEDGMENTS
REFERENCES

---

approach has been successfully applied to larger examples; some of these are mentioned in Section 8. Section 1 describes how a system can be viewed in terms of a state machine, clients, and output devices. Coping with failures is the subject of Sections 2 to 5. An important class of optimizations—based on the use of time—is discussed in Section 6. Section 7 describes dynamic reconfiguration. The history of the approach and related work are discussed in Section 8.

### 1. STATE MACHINES

Services, servers, and most programming language structures for supporting modularity define state machines. A *state machine* consists of *state variables*, which encode its state, and *commands*, which transform its state. Each command is implemented by a deterministic program; execution of the command is atomic with respect to other commands and modifies the state variables and/or produces some output. A client of the state machine makes a request to execute a command. The request names a state machine, names the command to be performed, and contains any information needed by the command.

Output from request processing can be to an actuator (e.g., in a process-control system), to some other peripheral device (e.g., a disk or terminal), or to clients awaiting responses from prior requests.

In this tutorial, we will describe a state machine simply by listing its state variables and commands. As an example, state machine *memory* of Figure 1 implements a time-varying mapping from locations to values. A *read* command permits a client to determine the value currently associated with a location, and a *write* command associates a new value with a location.

For generality, our descriptions of state machines deliberately do not specify how command invocation is implemented. Commands might be implemented in any of the following ways:

- Using a collection of procedures that share data and are invoked by a **call**, as in a monitor.
- Using a single process that awaits messages containing requests and performs the actions they specify, as in a server.
- Using a collection of interrupt handlers, in which case a request is made by causing an interrupt, as in an operating system kernel. (Disabling interrupts permits each command to be executed to completion before the next is started.)

For example, the state machine of Figure 2 implements commands to ensure that at all times at most one client has been granted access to some resource. In it,  $x \circ y$  denotes the result of appending  $y$  to the end of list  $x$ ,  $head(x)$  denotes the first element of list  $x$ , and  $tail(x)$  denotes the list obtained by deleting the first element of list  $x$ . This state machine would probably be implemented as part of the supervisor-call handler of an operating system kernel.

Requests are processed by a state machine one at a time, in an order that is consistent with potential causality. Therefore, clients of a state machine can make the following assumptions about the order in which requests are processed:

- O1: Requests issued by a single client to a given state machine  $sm$  are

```

memory: state_machine
  var store:array[0..n] of word
  read: command(loc:0..n)
    send store[loc] to client
    end read;
  write: command(loc:0..n, value:word)
    store[loc]:=value
    end write
  end memory

```

Figure 1. A memory.

```

mutex: state_machine
  var user:client_id init Φ;
      waiting:list of client_id init Φ;
  acquire: command
    if user = Φ → send OK to client;
      user := client
    □ user ≠ Φ → waiting := waiting ∘ client
    fi
    end acquire
  release: command
    if waiting = Φ → user := Φ
    □ waiting ≠ Φ → send OK to head(waiting);
      user := head(waiting);
      waiting := tail(waiting)
    fi
    end release
  end mutex

```

Figure 2. A resource allocator.

processed by  $sm$  in the order they were issued.

- O2: If the fact that request  $r$  was made to a state machine  $sm$  by client  $c$  could have caused a request  $r'$  to be made by a client  $c'$  to  $sm$ , then  $sm$  processes  $r$  before  $r'$ .

Note that due to communications network delays, O1 and O2 do not imply that a state machine will process requests in the order made or in the order received.

To keep our presentation independent of the interprocess communication mechanism used to transmit requests to state machines, we will program client requests as tuples of the form

$\langle state\_machine.command, arguments \rangle$

and postulate that any results from processing a request are returned using mes-

sages. For example, a client might execute

```

⟨memory.write, 100, 16.2⟩;
⟨memory.read, 100⟩;
receive v from memory

```

to set the value of location 100 to 16.2, request the value of location 100, and await that value, setting  $v$  to it upon receipt.

The defining characteristic of a state machine is not its syntax but that it specifies a deterministic computation that reads a stream of requests and processes each, occasionally producing output:

**Semantic Characterization of a State Machine.** Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.

Not all collections of commands necessarily satisfy this characterization. Consider the following program to solve a simple process-control problem in which an actuator is adjusted repeatedly based on the value of a sensor. Periodically, a client reads a sensor, communicates the value read to state machine  $pc$ , and delays approximately  $D$  seconds:

```

monitor:
  process
    do true → val := sensor;
      ⟨pc.adjust, val⟩;
      delay D
    od
  end monitor

```

State machine  $pc$  adjusts an actuator based on past adjustments saved in state variable  $q$ , the sensor reading, and a control function  $F$ :

```

pc: state_machine
  var q:real;

  adjust:
    command(sensor_val:real)
      q := F(q, sensor_val);
      send q to actuator
    end adjust
  end pc

```

Although it is tempting to structure  $pc$  as a single command that loops—reading from the sensor, evaluating  $F$ , and writing to  $actuator$ —if the value of the sensor is

time varying, then the result would not satisfy the semantic characterization given above and therefore would not be a state machine. This is because values sent to *actuator* (the output of the state machine) would not depend solely on the requests made to the state machine but would, in addition, depend on the execution speed of the loop. In the structure used above, this problem has been avoided by moving the loop into *monitor*.

In practice, having to structure a system in terms of state machines and clients does not constitute a real restriction. Anything that can be structured in terms of procedures and procedure calls can also be structured using state machines and clients—a state machine implements the procedure, and requests implement the procedure calls. In fact, state machines permit more flexibility in system structure than is usually available with procedure calls. With state machines, a client making a request is not delayed until that request is processed, and the output of a request can be sent someplace other than to the client making the request. We have not yet encountered an application that could not be programmed cleanly in terms of state machines and clients.

## 2. FAULT TOLERANCE

Before turning to the implementation of fault-tolerant state machines, we must introduce some terminology concerning failures. A component is considered *faulty* once its behavior is no longer consistent with its specification. In this paper, we consider two representative classes of faulty behavior:

**Byzantine Failures.** The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components [Lamport et al. 1982].

**Fail-stop Failures.** In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops [Schneider 1984].

Byzantine failures can be the most disruptive, and there is anecdotal evidence that such failures do occur in practice. Allowing

Byzantine failures is the weakest possible assumption that could be made about the effects of a failure. Since a design based on assumptions about the behavior of faulty components runs the risk of failing if these assumptions are not satisfied, it is prudent that life-critical systems tolerate Byzantine failures. For most applications, however, it suffices to assume fail-stop failures.

A system consisting of a set of distinct components is *t fault tolerant* if it satisfies its specification provided that no more than *t* of those components become faulty during some interval of interest.<sup>2</sup> Fault-tolerance traditionally has been specified in terms of mean time between failures (MTBF), probability of failure over a given interval, and other statistical measures [Siewiorek and Swarz 1982]. Although it is clear that such characterizations are important to the users of a system, there are advantages in describing fault tolerance of a system in terms of the maximum number of component failures that can be tolerated over some interval of interest. Asserting that a system is *t* fault tolerant makes explicit the assumptions required for correct operation; MTBF and other statistical measures do not. Moreover, *t* fault tolerance is unrelated to the reliability of the components that make up the system and therefore is a measure of the fault tolerance supported by the system architecture, in contrast to fault tolerance achieved simply by using reliable components. MTBF and other statistical reliability measures of a *t* fault-tolerant system can be derived from statistical reliability measures for the components used in constructing that system—in particular, the probability that there will be *t* or more failures during the operating interval of interest. Thus, *t* is typically chosen based on statistical measures of component reliability.

## 3. FAULT-TOLERANT STATE MACHINES

A *t* fault-tolerant version of a state machine can be implemented by replicating that

---

<sup>2</sup> A *t* fault-tolerant system might continue to operate correctly if more than *t* failures occur, but correct operation cannot be guaranteed.

state machine and running a replica on each of the processors in a distributed system. Provided each replica being run by a nonfaulty processor starts in the same initial state and executes the same requests in the same order, then each will do the same thing and produce the same output. Thus, if we assume that each failure can affect at most one processor, hence one state machine replica, then by combining the output of the state machine replicas of this *ensemble*, we can obtain the output for the  $t$  fault-tolerant state machine.

When processors can experience Byzantine failures, an ensemble implementing a  $t$  fault-tolerant state machine must have at least  $2t + 1$  replicas, and the output of the ensemble is the output produced by the majority of the replicas. This is because with  $2t + 1$  replicas, the majority of the outputs remain correct even after as many as  $t$  failures. If processors experience only fail-stop failures, then an ensemble containing  $t + 1$  replicas suffices, and the output of the ensemble can be the output produced by any of its members. This is because only correct outputs are produced by fail-stop processors, and after  $t$  failures one nonfaulty replica will remain among the  $t + 1$  replicas.

The key, then, for implementing a  $t$  fault-tolerant state machine is to ensure the following:

**Replica Coordination.** All replicas receive and process the same sequence of requests.

This can be decomposed into two requirements concerning dissemination of requests to replicas in an ensemble.

**Agreement.** Every nonfaulty state machine replica receives every request.

**Order.** Every nonfaulty state machine replica processes the requests it receives in the same relative order.

Notice that Agreement governs the behavior of a client in interacting with state machine replicas and that Order governs the behavior of a state machine replica with respect to requests from various clients. Thus, although Replica Coordination could

be partitioned in other ways, the Agreement–Order partitioning is a natural choice because it corresponds to the existing separation of the client from the state machine replicas.

Implementations of Agreement and Order are discussed in Sections 3.1 and 3.2. These implementations make no assumptions about clients or commands. Although this generality is useful, knowledge of commands allows Replica Coordination, hence Agreement and Order, to be weakened and thus allows cheaper protocols to be used for managing the replicas in an ensemble. Examples of two common weakenings follow.

First, Agreement can be relaxed for read-only requests when fail-stop processors are being assumed. When processors are fail stop, a request  $r$  whose processing does not modify state variables need only be sent to a single nonfaulty state machine replica. This is because the response from this replica is—by definition—guaranteed to be correct and because  $r$  changes no state variables, the state of the replica that processes  $r$  will remain identical to the states of replicas that do not.

Second, Order can be relaxed for requests that commute. Two requests  $r$  and  $r'$  *commute* in a state machine  $sm$  if the sequence of outputs and final state of  $sm$  that would result from processing  $r$  followed by  $r'$  is the same as would result from processing  $r'$  followed by  $r$ . An example of a state machine where Order can be relaxed appears in Figure 3. State machine *tally* determines which from among a set of alternatives receives at least  $MAJ$  votes and sends this choice to *SYSTEM*. If clients cannot vote more than once and the number of clients  $Cno$  satisfies  $2MAJ > Cno$ , then every request commutes with every other. Thus, implementing Order would be unnecessary—different replicas of the state machine will produce the same outputs even if they process requests in different orders. On the other hand, if clients can vote more than once or  $2MAJ \leq Cno$ , then reordering requests might change the outcome of the election.

Theories for constructing state machine ensembles that do not satisfy Replica Coordination are proposed in Aizikowitz

```

tally: state_machine
  var votes: array[candidate] of integer init 0
  cast_vote: command(choice:candidate)
    votes[choice] := votes[choice] + 1;
    if votes[choice] ≥ MAJ → send choice to
      SYSTEM;
      halt
    □ votes[choice] < MAJ → skip
    fi
    end cast_vote
  end tally

```

Figure 3. Election.

[1989] and Mancini and Pappalardo [1988]. Both theories are based on proving that an ensemble of state machines implements the same specification as a single replica does. The approach taken in Aizikowitz [1989] uses temporal logic descriptions of state sequences, whereas the approach in Mancini and Pappalardo [1988] uses an algebra of action sequences. A detailed description of this work is beyond the scope of this tutorial.

### 3.1 Agreement

The Agreement requirement can be satisfied by using any protocol that allows a designated processor, called the *transmitter*, to disseminate a value to some other processors in such a way that

- IC1: All nonfaulty processors agree on the same value.
- IC2: If the transmitter is nonfaulty, then all nonfaulty processors use its value as the one on which they agree.

Protocols to establish IC1 and IC2 have received considerable attention in the literature and are sometimes called *Byzantine Agreement protocols*, *reliable broadcast protocols*, or simply *agreement protocols*. The hard part in designing such protocols is coping with a transmitter that fails part way through an execution. See Strong and Dolev [1983] for protocols that can tolerate Byzantine processor failures and Schneider et al. [1984] for a (significantly cheaper) protocol that can tolerate (only) fail-stop processor failures.

If requests are distributed to all state machine replicas by using a protocol that

satisfies IC1 and IC2, then the Agreement requirement is satisfied. Either the client can serve as the transmitter or the client can send its request to a single state machine replica and let that replica serve as the transmitter. When the client does not itself serve as the transmitter, however, the client must ensure that its request is not lost or corrupted by the transmitter before the request is disseminated to the state machine replicas. One way to monitor for such corruption is by having the client be among the processors that receive the request from the transmitter.

### 3.2 Order and Stability

The Order requirement can be satisfied by assigning unique identifiers to requests and having state machine replicas process requests according to a total ordering relation on these unique identifiers. This is equivalent to requiring the following, where a request is defined to be *stable* at  $sm_i$  once no request from a correct client and bearing a lower unique identifier can be subsequently delivered to state machine replica  $sm_i$ :

**Order Implementation.** A replica next processes the stable request with the smallest unique identifier.

Further refinement of Order Implementation requires selecting a method for assigning unique identifiers to requests and devising a stability test for that assignment method. Note that any method for assigning unique identifiers is constrained by O1 and O2 of Section 1, which imply that if request  $r_i$  could have caused request  $r_j$  to be made then  $uid(r_i) < uid(r_j)$  holds, where  $uid(r)$  is the unique identifier assigned to a request  $r$ .

In the sections that follow, we give three refinements of the Order Implementation. Two are based on the use of clocks; a third uses an ordering defined by the replicas of the ensemble.

#### 3.2.1 Using Logical Clocks

A *logical clock* [Lamport 1978a] is a mapping  $\hat{T}$  from events to the integers.  $\hat{T}(e)$ ,

the “time” assigned to an event  $e$  by logical clock  $\hat{T}$ , is an integer such that for any two distinct events  $e$  and  $e'$ , either  $\hat{T}(e) < \hat{T}(e')$  or  $\hat{T}(e) > \hat{T}(e')$ , and if  $e$  might be responsible for causing  $e'$  then  $\hat{T}(e) < \hat{T}(e')$ . It is a simple matter to implement logical clocks in a distributed system. Associated with each process  $p$  is a counter  $\hat{T}_p$ . In addition, a *timestamp* is included in each message sent by  $p$ . This timestamp is the value of  $\hat{T}_p$  when that message is sent.  $\hat{T}_p$  is updated according to the following:

- LC1:  $\hat{T}_p$  is incremented after each event at  $p$ .
- LC2: Upon receipt of a message with timestamp  $\tau$ , process  $p$  resets  $\hat{T}_p$ :

$$\hat{T}_p := \max(\hat{T}_p, \tau) + 1.$$

The value of  $\hat{T}(e)$  for an event  $e$  that occurs at processor  $p$  is constructed by appending a fixed-length bit string that uniquely identifies  $p$  to the value of  $\hat{T}_p$  when  $e$  occurs.

Figure 4 illustrates the use of this scheme for implementing logical clocks in a system of three processors,  $p$ ,  $q$ , and  $r$ . Events are depicted by dots, and an arrow is drawn between events  $e$  and  $e'$  if  $e$  might be responsible for causing event  $e'$ . For example, an arrow between events in different processes starts from the event corresponding to the sending of a message and ends at the event corresponding to the receipt of that message. The value of  $\hat{T}_p(e)$  for each event  $e$  is written above that event.

If  $\hat{T}(e)$  is used as the unique identifier associated with a request whose issuance corresponds to event  $e$ , the result is a total ordering on the unique identifiers that satisfies O1 and O2. Thus, a logical clock can be used as the basis of an Order Implementation if we can formulate a way to determine when a request is stable at a state machine replica.

It is pointless to implement a stability test in a system in which Byzantine failures are possible and a processor or message can be delayed for an arbitrary length of time without being considered faulty. This is because no deterministic protocol can implement agreement under these conditions [Fischer et al. 85].<sup>3</sup> Since it is impossible to satisfy the Agreement requirement, there is no point in satisfying the Order require-

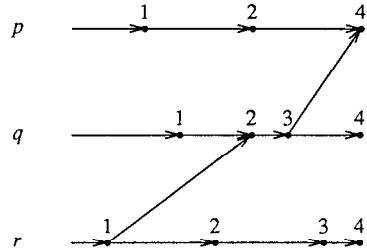


Figure 4. Logical clock example.

ment. The case in which relative speeds of nonfaulty processors and messages is bounded is equivalent to assuming that they have synchronized real-time clocks and will be considered shortly. This leaves the case in which fail-stop failures are possible and a process or message can be delayed for an arbitrary length of time without being considered faulty. Thus, we now turn to devising a stability test for that environment.

By attaching sequence numbers to the messages between every pair of processors, it is trivial to ensure the following property holds of communications channels:

**FIFO Channels.** Messages between a pair of processors are delivered in the order sent.

For fail-stop processors, we can also assume the following:

**Failure Detection Assumption.** A processor  $p$  detects that a fail-stop processor  $q$  has failed only after  $p$  has received the last message sent to  $p$  by  $q$ .

The Failure Detection Assumption is consistent with FIFO Channels, since the failure event for a fail-stop processor necessarily happens after the last message sent by the processor and, therefore, should be received after all other messages.

Under these two assumptions, the following stability test can be used:

**Logical Clock Stability Test Tolerating Fail-stop Failures.** Every client

<sup>3</sup> The result of Fischer et al. [1985] is actually stronger than this. It states that IC1 and IC2 cannot be achieved by a deterministic protocol in an asynchronous system with a single processor that fails in an even less restrictive manner—by simply halting.

periodically makes some—possibly null—request to the state machine. A request is stable at replica  $sm_i$  if a request with larger timestamp has been received by  $sm_i$  from every client running on a nonfaulty processor.

To see why this stability test works, we show that once a request  $r$  is stable at  $sm_i$ , no request with smaller unique identifier (timestamp) will be received. First, consider clients that  $sm_i$  does not detect as being faulty. Because logical clocks are used to generate unique identifiers, any request made by a client  $c$  must have a larger unique identifier than was assigned to any previous request made by  $c$ . Therefore, from the FIFO Channels assumption, we conclude that once a request from a nonfaulty client  $c$  is received by  $sm_i$ , no request from  $c$  with a smaller unique identifier than  $uid(r)$  can be received by  $sm_i$ . This means that once requests with larger unique identifiers than  $uid(r)$  have been received from every nonfaulty client, it is not possible to receive a request with a smaller unique identifier than  $uid(r)$  from these clients. Next, for a client  $c$  that  $sm_i$  detects as faulty, the Failure Detection Assumption implies that no request from  $c$  will be received by  $sm_i$ . Thus, once a request  $r$  is stable at  $sm_i$ , no request with a smaller timestamp can be received from a client—faulty or nonfaulty.

### 3.2.2 Synchronized Real-Time Clocks

A second way to produce unique request identifiers satisfying O1 and O2 is by using approximately synchronized real-time clocks.<sup>4</sup> Define  $T_p(e)$  to be the value of the real-time clock at processor  $p$  when event  $e$

<sup>4</sup> A number of protocols to achieve clock synchronization while tolerating Byzantine failures have been proposed [Halpern et al. 1984; Lamport and Melliar-Smith 1984]. See Schneider [1986] for a survey. The protocols all require that known bounds exist for the execution speed and clock rates of nonfaulty processors and for message delivery delays along nonfaulty communications links. In practice, these requirements do not constitute a restriction. Clock synchronization achieved by the protocols is proportional to the variance in message delivery delay, making it possible to satisfy the restriction—necessary to ensure O2—that message delivery delay exceeds clock synchronization.

occurs. We can use  $T_p(e)$  followed by a fixed-length bit string that uniquely identifies  $p$  as the unique identifier associated with a request made as event  $e$  by a client running on a processor  $p$ . To ensure that O1 and O2 (of Section 1) hold for unique identifiers generated in this manner, two restrictions are required. O1 follows provided no client makes two or more requests between successive clock ticks. Thus, if processor clocks have a resolution of  $R$  seconds, then each client can make at most one request every  $R$  seconds. O2 follows provided the degree of clock synchronization is better than the minimum message delivery time. In particular, if clocks on different processors are synchronized to within  $\delta$  seconds, then it must take more than  $\delta$  seconds for a message from one client to reach another. Otherwise, O2 would be violated because a request  $r$  made by the one client could have a unique identifier that was smaller than a request  $r'$  made by another, even though  $r$  was caused by a message sent after  $r'$  was made.

When unique request identifiers are obtained from synchronized real-time clocks, a stability test can be implemented by exploiting these clocks and the bounds on message delivery delays. Define  $\Delta$  to be constant such that a request  $r$  with unique identifier  $uid(r)$  will be received by every correct processor no later than time  $uid(r) + \Delta$  according to the local clock at the receiving processor. Such a  $\Delta$  must exist if requests are disseminated using a protocol that employs a fixed number of rounds, like the ones cited above for establishing IC1 and IC2.<sup>5</sup> By definition, once the clock on a processor  $p$  reaches time  $\tau$ ,  $p$  cannot subsequently receive a request  $r$  such that  $uid(r) < \tau - \Delta$ . Therefore, we have the following stability test:

**Real-time Clock Stability Test Tolerating Byzantine Failures I.** A request  $r$  is stable at a state machine replica  $sm_i$

<sup>5</sup> In general,  $\Delta$  will be a function of the variance in message delivery delay, the maximum message delivery delay, and the degree of clock synchronization. See Cristian et al. [1985] for a detailed derivation for  $\Delta$  in a variety of environments.

being executed by processor  $p$  if the local clock at  $p$  reads  $\tau$  and  $uid(r) < \tau - \Delta$ .

One disadvantage of this stability test is that it forces the state machine to lag behind its clients by  $\Delta$ , where  $\Delta$  is proportional to the worst-case message delivery delay. This disadvantage can be avoided. Due to property O1 of the total ordering on request identifiers, if communications channels satisfy FIFO Channels, then a state machine replica that has received a request  $r$  from a client  $c$  can subsequently receive from  $c$  only requests with unique identifiers greater than  $uid(r)$ . Thus, a request  $r$  is also stable at a state machine replica provided a request with a larger unique identifier has been received from every client.

**Real-time Clock Stability Test Tolerating Byzantine Failures II.** A request  $r$  is stable at a state machine replica  $sm_i$  if a request with a larger unique identifier has been received from every client.

This second stability test is never passed if a (faulty) processor refuses to make requests. However, by combining the first and second test so that a request is considered stable when it satisfies either test, a stability test results that lags clients by  $\Delta$  only when faulty processors or network delays force it. Such a combined test is discussed in [Gopal et al. 1990].

### 3.2.3 Using Replica-Generated Identifiers

In the previous two refinements of the Order Implementation, clients determine the order in which requests are processed—the unique identifier  $uid(r)$  for a request  $r$  is assigned by the client making that request. In the following refinement of the Order Implementation, the state machine replicas determine this order. Unique identifiers are computed in two phases. In the first phase, which can be part of the agreement protocol used to satisfy the Agreement requirement, state machine replicas propose candidate unique identifiers for a request. Then, in the second phase, one of these candidates is selected and it becomes the unique identifier for that request.

The advantage of this approach to computing unique identifiers is that communication between all processors in the system is not necessary. When logical clocks or synchronized real-time clocks are used in computing unique request identifiers, all processors hosting clients or state machine replicas must communicate. In the case of logical clocks, this communication is needed in order for requests to become stable; in the case of synchronized real-time clocks, this communication is needed in order to keep the clocks synchronized.<sup>6</sup> In the replica-generated identifier approach of this section, the only communication required is among processors running the client and state machine replicas.

By constraining the possible candidates proposed in phase 1 for a request's unique identifier, it is possible to obtain a simple stability test. To describe this stability test, some terminology is required. We say that a state machine replica  $sm_i$  has *seen* a request  $r$  once  $sm_i$  has received  $r$  and proposed a candidate unique identifier for  $r$ . We say that  $sm_i$  has *accepted*  $r$  once that replica knows the ultimate choice of unique identifier for  $r$ . Define  $cuid(sm_i, r)$  to be the candidate unique identifier proposed by replica  $sm_i$  for request  $r$ . Two constraints that lead to a simple stability test are:

UID1:  $cuid(sm_i, r) \leq uid(r)$ .

UID2: If a request  $r'$  is seen by replica  $sm_i$  after  $r$  has been accepted by  $sm_i$  then  $uid(r) < cuid(sm_i, r')$ .

If these constraints hold throughout execution, then the following test can be used to determine whether a request is stable at a state machine replica:

**Replica-Generated Identifiers Stability Test.** A request  $r$  that has been accepted by  $sm_i$  is stable provided there is no

---

<sup>6</sup>This communications cost argument illustrates an advantage of having a client forward its request to a single state machine replica that then serves as the transmitter for disseminating the request. In effect, that state machine replica becomes the client of the state machine, and so communication need only involve those processors running state machine replicas.

request  $r'$  that has (i) been seen by  $sm_i$ , (ii) not been accepted by  $sm_i$ , and (iii) for which  $cuid(sm_i, r') \leq uid(r)$  holds.

To prove that this stability test works, we must show that once an accepted request  $r$  is deemed stable at  $sm_i$ , no request with a smaller unique identifier will be subsequently accepted at  $sm_i$ . Let  $r$  be a request that, according to the Replica-Generated Identifiers Stability Test, is stable at replica  $sm_i$ . Due to UID2, for any request  $r'$  that has not been seen by  $sm_i$ ,  $uid(r) < cuid(sm_i, r')$  holds. Thus, by transitivity using UID1,  $uid(r) < uid(r')$  holds, and we conclude that  $r'$  cannot have a smaller unique identifier than  $r$ . Now consider the case in which request  $r'$  has been seen but not accepted by  $sm_i$ , and—because the stability test for  $r$  is satisfied— $uid(r) < cuid(sm_i, r')$  holds. Due to UID1, we conclude that  $uid(r) < uid(r')$  holds and, therefore,  $r'$  does not have a smaller unique identifier than  $r$ . Thus, we have shown that once a request  $r$  satisfies the Replica-Generated Identifiers Stability Test at  $sm_i$ , any request  $r'$  that is accepted by  $sm_i$  will satisfy  $uid(r) < uid(r')$ , as desired.

Unlike clock-generated unique identifiers for requests, replica-generated ones do not necessarily satisfy O1 and O2 of Section 1. Without further restrictions, it is possible for a client to make a request  $r$ , send a message to another client causing request  $r'$  to be issued, yet have  $uid(r') < uid(r)$ . However, O1 and O2 will hold provided that once a client starts disseminating a request to the state machine replicas, the client performs no other communication until every state machine replica has accepted that request. To see why this works, consider a request  $r$  being made by some client and suppose some request  $r'$  was influenced by  $r$ . The delay ensures that  $r$  is accepted by every state machine replica  $sm_j$  before  $r'$  is seen. Thus, from UID2 we conclude  $uid(r) < cuid(sm_i, r')$  and, by transitivity with UID1, that  $uid(r) < uid(r')$ , as required.

To complete this Order Implementation, we have only to devise protocols for computing unique identifiers and candidate

unique identifiers such that:

$$\bullet \text{ UID1 and UID2 are satisfied.} \quad (1)$$

$$\bullet r \neq r' \Rightarrow uid(r) \neq uid(r'). \quad (2)$$

$$\bullet \text{ Every request that is seen eventually becomes accepted.} \quad (3)$$

One simple solution for a system of fail-stop processors is the following:

**Replica-generated Unique Identifiers.**

In a system with  $N$  clients, each state machine replica  $sm_i$  maintains two variables:

$SEEN_i$  is the largest  $cuid(sm_i, r)$  assigned to any request  $r$  so far seen by  $sm_i$ , and

$ACCEPT_i$  is the largest  $uid(r)$  assigned to any request  $r$  so far accepted by  $sm_i$ .

Upon receipt of a request  $r$ , each replica  $sm_i$  computes

$$cuid(sm_i, r) :=$$

$$\max(\lfloor SEEN_i \rfloor, \lfloor ACCEPT_i \rfloor)$$

$$+ 1 + i/N. \quad (4)$$

(Notice, this means that all candidate unique identifiers are themselves unique.) The replica then disseminates (using an agreement protocol)  $cuid(sm_i, r)$  to all other replicas and awaits receipt of a candidate unique identifier for  $r$  from every nonfaulty replica, participating in the agreement protocol for that value as well. Let  $NF$  be the set of replicas from which candidate unique identifiers were received. Finally, the replica computes

$$uid(r) := \max_{sm_j \in NF} (cuid(sm_j, r)) \quad (5)$$

and accepts  $r$ .

We prove that this protocol satisfies (1)–(3) as follows. UID1 follows from using assignment (5) to compute  $uid(r)$ , and UID2 follows from assignment (4) to compute  $cuid(sm_i, r)$ . To conclude that (2) holds, we argue as follows. Because an agreement protocol is used to disseminate candidate unique identifiers, all replicas receive the same values from the same replicas. Thus, all replicas will execute the same assignment statement (5), and all will compute the same value for  $uid(r)$ . To establish

that these  $uid(r)$  values are unique for each request, it suffices to observe that maximums of disjoint subsets of a collection of unique values—the candidate unique identifiers—are also unique. Finally, to establish (3), that every request that is seen is eventually accepted, we must prove that for each replica  $sm_j$ , a replica  $sm_i$  eventually learns  $cuid(sm_j, r)$  or learns that  $sm_j$  has failed. This follows trivially from the use of an agreement protocol to distribute the  $cuid(sm_j, r)$  and the definition of a fail-stop processor.

An optimization of our Replica-generated Unique Identifiers protocol is the basis for the ABCAST protocol in the ISIS Toolkit [Birman and Joseph 1987] developed at Cornell. In this optimization, candidate unique identifiers are returned to the client instead of being disseminated to the other state machine replicas. The client then executes assignment (5) to compute  $uid(r)$ . Finally, an agreement protocol is used by the client in disseminating  $uid(r)$  to the state machine replicas. Some unique replica takes over for the client if the client fails.

It is possible to modify our Replica-generated Unique Identifiers protocol for use in systems where processors can exhibit Byzantine failures, have synchronized real-time clocks, and communications channels have bounded message-delivery delays—the same environment as was assumed for using synchronized real-time clocks to generate unique identifiers. The following changes are required. First, each replica  $sm_i$  uses timeouts so that  $sm_i$  cannot be forever delayed waiting to receive and participate in the agreement protocol for disseminating a candidate unique identifier from a faulty replica  $sm_j$ . Second, if  $sm_i$  does determine that  $sm_j$  has timed out,  $sm_i$  disseminates “ $sm_j$  timeout” to all replicas (by using an agreement protocol). Finally,  $NF$  is the set of replicas in the ensemble less any  $sm_j$  for which “ $sm_j$  timeout” has been received from  $t + 1$  or more replicas. Notice, Byzantine failures that cause faulty replicas to propose candidate unique identifiers not produced by (4) do not cause difficulty. This is because candidate unique identifiers that

are too small have no effect on the outcome of (5) at nonfaulty replicas and those that are too large will satisfy UID1 and UID2.

#### 4. TOLERATING FAULTY OUTPUT DEVICES

It is not possible to implement a  $t$  fault-tolerant system by using a single voter to combine the outputs of an ensemble of state machine replicas into one output. This is because a single failure—of the voter—can prevent the system from producing the correct output. Solutions to this problem depend on whether the output of the state machine implemented by the ensemble is to be used within the system or outside the system.

##### 4.1 Outputs Used Outside the System

If the output of the state machine is sent to an output device, then that device is already a single component whose failure cannot be tolerated. Thus, being able to tolerate a faulty voter is not sufficient—the system must also be able to tolerate a faulty output device. The usual solution to this problem is to replicate the output device and voter. Each voter combines the output of each state machine replica, producing a signal that drives one output device. Whatever reads the outputs of the system is assumed to combine the outputs of the replicated devices. This reader, which is not considered part of the computing system, implements the critical voter.

If output devices can exhibit Byzantine failures, then by taking the output produced by the majority of the devices,  $2t + 1$ -fold replication permits up to  $t$  faulty output devices to be tolerated. For example, a flap on an airplane wing might be designed so that when the  $2t + 1$  actuators that control it do not agree, the flap always moves in the direction of the majority (rather than twisting). If output devices exhibit only fail-stop failures, then only  $t + 1$ -fold replication is necessary to tolerate  $t$  failures because any output produced by a fail-stop output device can be assumed correct. For example, video display terminals usually present information with

enough redundancy so that they can be treated as fail stop—failure detection is implemented by the viewer. With such an output device, a human user can look at one of  $t + 1$  devices, decide whether the output is faulty, and only if it is faulty, look at another, and so on.

#### 4.2 Outputs Used Inside the System

If the output of the state machine is to a client, then the client itself can combine the outputs of state machine replicas in the ensemble. Here, the voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant. When Byzantine failures are possible, the client waits until it has received  $t + 1$  identical responses, each from a different member of the ensemble, and takes that as the response from the  $t$  fault-tolerant state machine. When only fail-stop failures are possible, the client can proceed as soon as it has received a response from any member of the ensemble, since any output produced by a replica must be correct.

When the client is executed on the same processor as one of the state machine replicas, optimization of client-implemented voting is possible.<sup>7</sup> This is because correctness of the processor implies that both the state machine replica and client will be correct. Therefore, the response produced by the state machine replica running locally can be used as that client's response from the  $t$  fault-tolerant state machine. And, if the processor is faulty, we are entitled to view the client as being faulty, so it does not matter what state machine responses the client receives. Summarizing, we have the following:

**Dependent-Failures Output Optimization.** If a client and a state machine replica run on the same processor, then even when

---

<sup>7</sup> Care must be exercised when analyzing the fault tolerance of such a system because a single processor failure can now cause two system components to fail. Implicit in most of our discussions is that system components fail independently. It is not always possible to transform a  $t$  fault-tolerant system in which clients and state machine replicas have independent failures to one in which they share processors.

Byzantine failures are possible, the client need not gather a majority of responses to its requests to the state machine. It can use the single response produced locally.

### 5. TOLERATING FAULTY CLIENTS

Implementing a  $t$  fault-tolerant state machine is not sufficient for implementing a  $t$  fault-tolerant system. Faults might result in clients making requests that cause the state machine to produce erroneous output or that corrupt the state machine so that subsequent requests from nonfaulty clients are incorrectly processed. Therefore, in this section we discuss various methods for insulating the state machine from faults that affect clients.

#### 5.1 Replicating the Client

One way to avoid having faults affect a client is by replicating the client and running each replica on hardware that fails independently. This replication, however, also requires changes to state machines that handle requests from that client. This is because after a client has been replicated  $N$ -fold, any state machine it interacts with receives  $N$  requests—one from each client replica—when it formerly receives a single request. Moreover, corresponding requests from different client replicas will not necessarily be identical. First, they will differ in their unique identifiers. Second, unless the original client is itself a state machine and the methods of Section 3 are used to coordinate the replicas, corresponding requests from different replicas can also differ in their content. For example, if a client makes requests based on the value of some time-varying sensor, then its replicas will each read their sensors at slightly different times and, therefore, make different requests.

We first consider modifications to a state machine  $sm$  for the case in which requests from different client replicas are known to differ only in their unique identifiers. For this case, modifications are needed for coping with receiving  $N$  requests instead of a single one. These modifications involve changing each command so that instead of processing every request received, requests

are buffered until enough<sup>8</sup> have been received; only then is the corresponding command performed (a single time). In effect, a voter is being added to  $sm$  to control invocation of its commands. Client replication can be made invisible to the designer of a state machine by including such a voter in the support software that receives requests, tests for stability, and orders stable requests by unique identifier.

Modifying the state machine for the case in which requests from different client replicas can also differ in their content typically requires exploiting knowledge of the application. As before, the idea is to transform multiple requests into a single one. For example, in a  $t$  fault-tolerant system, if  $2t + 1$  different requests are received, each containing the value of a sensor, then a single request containing the median of those values might be constructed and processed by the state machine. (Given at most  $t$  Byzantine faults, the median of  $2t + 1$  values is a reasonable one to use because it is bounded from above and below by a nonfaulty value.) A general method for transforming multiple requests containing sensor values into a single request is discussed in Marzullo [1989]. That method is based on viewing a sensor value as an interval that includes the actual value being measured; a single interval (sensor) is computed from a set of intervals by using a fault-tolerant intersection algorithm.

## 5.2 Defensive Programming

Sometimes a client cannot be made fault tolerant by using replication. In some circumstances, due to the unavailability of sensors or processors, it simply might not be possible to replicate the client. In other circumstances, the application semantics might not afford a reasonable way to transform multiple requests from client replicas into the single request needed by the state machine. In all of these circumstances, careful design of state machines can limit

<sup>8</sup>If Byzantine failures are possible, then a  $t$  fault-tolerant client requires  $2t + 1$ -fold replication and a command is performed after  $t + 1$  requests have been received. If failures are restricted to fail stop, then  $t + 1$ -fold replication will suffice, and a command can be performed after a single request has been received.

the effects of requests from faulty clients. For example, *memory* (Figure 1) permits any client to write to any location. Therefore, a faulty client can overwrite all locations, destroying information. This problem could be prevented by restricting write requests from each client to only certain memory locations—the state machine can enforce this.

Including tests in commands is another way to design a state machine that cannot be corrupted by requests from faulty clients. For example, *mutex* as specified in Figure 2, will execute a *release* command made by any client—even one that does not have access to the resource. Consequently, a faulty client could issue such a request and cause *mutex* to grant a second client access to the resource before the first has relinquished access. A better formulation of *mutex* ignores *release* commands from all but the client to which exclusive access has been granted. This is implemented by changing the *release* in *mutex* to

```
release:
  command
    if user ≠ client → skip
    □ waiting = Φ ∧ user = client →
      user := Φ
    □ waiting ≠ Φ ∧ user = client →
      send OK to head(waiting);
      user := head(waiting);
      waiting := tail(waiting)
    fi
  end release
```

Sometimes, a faulty client *not* making a request can be just as catastrophic as one making an erroneous request. For example, if a client of *mutex* failed and stopped while it had exclusive access to the resource, then no client could be granted access to the resource. Of course, unless we are prepared to bound the length of time that a correctly functioning process can retain exclusive access to the resource, there is little we can do about this problem. This is because there is no way for a state machine to distinguish between a client that has stopped executing because it has failed and one that is executing very slowly. However, given an upper bound  $B$  on the interval between an *acquire* and the following *release*, the *acquire* command of *mutex* can automatically schedule *release* on behalf of a client.

We use the notation

**schedule**  $\langle REQUEST \rangle$  **for**  $+ \tau$

to specify scheduling  $\langle REQUEST \rangle$  with a unique identifier at least  $\tau$  greater than the identifier on the request being processed. Such a request is called a *timeout request* and becomes stable at some time in the future, according to the stability test being used for client-generated requests. Unlike requests from clients, requests that result from executing **schedule** need not be distributed to all state machine replicas of the ensemble. This is because each state machine replica will independently **schedule** its own (identical) copy of the request.

We can now modify *acquire* so that a *release* operation is automatically scheduled. In the code that follows, *TIME* is assumed to be a function that evaluates to the current time. Note that *mutex* might now process two *release* commands on behalf of a client that has acquired access to the resource: one command from the client itself and one generated by its *acquire* request. The new state variable *time\_granted*, however, ensures that superfluous *release* commands are ignored. The code is

```

acquire:
  command
  if user =  $\Phi \rightarrow$ 
    send OK to client;
    time_granted := TIME;
    schedule
       $\langle mutex.timeout, time\_granted \rangle$ 
    for  $+ B$ 
     $\square user \neq \Phi \rightarrow waiting := waiting \circ client$ 
  fi
  end acquire

timeout:
  command (when_granted:integer)
  if when_granted  $\neq$ 
    time_granted  $\rightarrow$  skip
   $\square waiting = \Phi \wedge when\_granted =$ 
    time_granted  $\rightarrow user := \Phi$ 
   $\square waiting \neq \Phi \wedge when\_granted =$ 
    time_granted  $\rightarrow$ 
    send OK to head(waiting);
    user := head(waiting);
    time_granted := TIME;
    waiting := tail(waiting)
  fi
end timeout

```

## 6. USING TIME TO MAKE REQUESTS

A client need not explicitly send a message to make a request. Not receiving a request can trigger execution of a command—in effect, allowing the passage of time to transmit a request from client to state machine [Lamport 1984]. Transmitting a request using time instead of messages can be advantageous because protocols that implement IC1 and IC2 can be costly both in total number of messages exchanged and in delay. Unfortunately, using time to transmit requests has only limited applicability since the client cannot specify parameter values.

The use of time to transmit a request was used in Section 5 when we revised the *acquire* command of *mutex* to foil clients that failed to release the resource. There, a *release* request was automatically scheduled by *acquire* on behalf of a client being granted the resource. A client transmits a *release* request to *mutex* simply by permitting *B* (logical clock or real-time clock) time units to pass. It is only to increase utilization of the shared resource that a client might use messages to transmit a *release* request to *mutex* before *B* time units have passed.

A more dramatic example of using time to transmit a request is illustrated in connection with *tally* of Figure 3. Assume that

- all clients and state machine replicas have (logical or real time) clocks synchronized to within  $\Gamma$ ,

and

- the election starts at time *Strt* and this is known to all clients and state machine replicas.

Using time, a client can cast a vote for a *default* by doing nothing; only when a client casts a vote different from its default do we require that it actually transmits a request message. Thus, we have:

**Transmitting a Default Vote.** If client has not made a request by time *Strt* +  $\Gamma$ , then a request with that client's default vote has been made.

Notice that the default need not be fixed nor even known at the time a vote is cast.

For example, the default vote could be “vote for the first client that any client casts a nondefault vote for.” In that case, the entire election can be conducted as long as one client casts a vote by using actual messages.<sup>9</sup>

## 7. RECONFIGURATION

An ensemble of state machine replicas can tolerate more than  $t$  faults if it is possible to remove state machine replicas running on faulty processors from the ensemble and add replicas running on repaired processors. (A similar argument can be made for being able to add and remove copies of clients and output devices.) Let  $P(\tau)$  be the total number of processors at time  $\tau$  that are executing replicas of some state machine of interest, and let  $F(\tau)$  be the number of them that are faulty. In order for the ensemble to produce the correct output, we must have

**Combining Condition:**  $P(\tau) - F(\tau) > Enuf$  for all  $0 \leq \tau$ , where

$$Enuf = \begin{cases} \frac{P(\tau)}{2} & \text{if Byzantine failures are possible.} \\ 0 & \text{if only fail-stop failures are possible.} \end{cases}$$

A processor failure may cause the Combining Condition to be violated by increasing  $F(\tau)$ , thereby decreasing  $P(\tau) - F(\tau)$ . When Byzantine failures are possible, if a faulty processor can be identified, then removing it from the ensemble decreases  $Enuf$  without further decreasing  $P(\tau) - F(\tau)$ ; this can keep the Combining Condition from being violated. When only fail-stop failures are possible, increasing the number of nonfaulty processors—by adding one that has been repaired—is the only way to keep the Combining Condition from being violated because increasing  $P(\tau)$  is the only way to ensure that  $P(\tau) - F(\tau) > 0$  holds. Therefore, provided the following conditions hold, it may be possible to maintain the Combining Condition forever

<sup>9</sup> Observe that if Byzantine failures are possible, then a faulty client can be elected. Such problems are always possible when voters do not have detailed knowledge about the candidates in an election.

and thus tolerate an unbounded total number of faults over the life of the system:

- F1: If Byzantine failures are possible, then state machine replicas being executed by faulty processors are identified and removed from the ensemble before the Combining Condition is violated by subsequent processor failures.
- F2: State machine replicas running on repaired processors are added to the ensemble before the Combining Condition is violated by subsequent processor failures.

F1 and F2 constrain the rates at which failures and repairs occur.

Removing faulty processors from an ensemble of state machines can also improve system performance. This is because the number of messages that must be sent to achieve agreement is usually proportional to the number of state machine replicas that must agree on the contents of a request. In addition, some protocols to implement agreement execute in time proportional to the number of processors that are faulty. Removing faulty processors clearly reduces both the message complexity and time complexity of such protocols.

Adding or removing a client from the system is simply a matter of changing the state machine so that henceforth it responds to or ignores requests from that client. Adding an output device is also straightforward—the state machine starts sending output to that device. Removing an output device from a system is achieved by *disabling* the device. This is done by putting the device in a state that prevents it from affecting the environment. For example, a CRT terminal can be disabled by turning off the brightness so that the screen can no longer be read; a hydraulic actuator controlling the flap on an airplane wing can be disabled by opening a cutoff valve so that the actuator exerts no pressure on that control surface. As suggested by these examples, however, it is not always possible to disable a faulty output device: Turning off the brightness might have no effect on the screen, and the cutoff valve might not work. Thus, there are systems in which no

more than a total of  $t$  actuator faults can be tolerated because faulty actuators cannot be disabled.

The *configuration* of a system structured in terms of a state machine and clients can be described using three sets: the clients  $C$ , the state machine replicas  $S$ , and the output devices  $O$ .  $S$  is used by the agreement protocol and therefore must be known to clients and state machine replicas. It can also be used by an output device to determine which **send** operations made by state machine replicas should be ignored.  $C$  and  $O$  are used by state machine replicas to determine from which clients requests should be processed and to which devices output should be sent. Therefore,  $C$  and  $O$  must be available to all state machine replicas.

Two problems must be solved to support changing the system configuration. First, the values of  $C$ ,  $S$ , and  $O$  must be available when required. Second, whenever a client, state machine replica, or output device is added to the configuration, the state of that *element* must be updated to reflect the current state of the system. These problems are considered in the following two sections.

### 7.1 Managing the Configuration

The configuration of a system can be managed using the state machine in that system. Sets  $C$ ,  $S$ , and  $O$  are stored in state variables and changed by commands. Each configuration is *valid* for a collection of requests—those requests  $r$  such that  $uid(r)$  is in the range defined by two successive configuration-change requests. Thus, whenever a client, state machine replica, or output device performs an action connected with processing  $r$ , it uses the configuration that is valid for  $r$ . This means that a configuration-change request must schedule the new configuration for some point far enough in the future so that clients, state machine replicas, and output devices all find out about the new configuration before it actually comes into effect.

There are various ways to make configuration information available to the clients and output devices of a system. (The information is already available to the state

machine.) One is for clients and output devices to query the state machine periodically for information about relevant pending configuration changes. Obviously, communication costs for this scheme are reduced if clients and output devices share processors with state machine replicas. Another way to make configuration information available is for the state machine to include information about configuration changes in messages it sends to clients and output devices in the course of normal processing. Doing this requires periodic communication between the state machine and clients and between the state machine and output devices.

Requests to change the configuration of the system are made by a failure/recovery detection mechanism. It is convenient to think of this mechanism as a collection of clients, one for each element of  $C$ ,  $S$ , or  $O$ . Each of these *configurators* is responsible for detecting the failure or repair of the single object it manages and, when such an event is detected, for making a request to alter the configuration. A configurator is likely to be part of an existing client or state machine replica and might be implemented in a variety of ways.

When elements are fail stop, a configurator need only check the failure-detection mechanism of that element. When elements can exhibit Byzantine failures, detecting failures is not always possible. When it is possible, a higher degree of fault tolerance can be achieved by reconfiguration. A nonfaulty configurator satisfies two safety properties:

- C1: Only a faulty element is removed from the configuration.
- C2: Only a nonfaulty element is added to the configuration.

A configurator that does nothing satisfies C1 and C2. Changing the configuration enhances faults tolerance only if F1 and F2 also hold. For F1 and F2 to hold, a configurator must also (1) detect faults and cause elements to be removed and (2) detect repairs and cause elements to be added. Thus, the degree to which a configurator enhances fault tolerance is directly related to the degree to which (1) and (2) are achieved.

Here, the semantics of the application can be helpful. For example, to infer that a client is faulty, a state machine can compare requests made by different clients or by the same client over a period of time. To determine that a processor executing a state machine replica is faulty, the state machine can monitor messages sent by other state machine replicas during execution of an agreement protocol. And, by monitoring aspects of the environment being controlled by actuators, a state machine replica might be able to determine that an output device is faulty. Some elements, such as processors, have internal failure-detection circuitry that can be read to determine whether that element is faulty or has been repaired and restarted. A configurator for such an element can be implemented by having the state machine periodically poll this circuitry.

In order to analyze the fault tolerance of a system that uses configurators, failure of a configurator can be considered equivalent to the failure of the element that the configurator manages. This is because with respect to the Combining Condition, removal of a nonfaulty element from the system or addition of a faulty one is the same as that element failing. Thus, in a  $t$  fault-tolerant system, the sum of the number of faulty configurators that manage nonfaulty elements and the number of faulty components with nonfaulty configurators must be bounded by  $t$ .

## 7.2 Integrating a Repaired Object

Not only must an element being added to a configuration be nonfaulty, it also must have the correct state so that its actions will be consistent with those of the rest of the system. Define  $e[r_i]$  to be the state that a non-faulty system element  $e$  should be in after processing requests  $r_0$  through  $r_i$ . An element  $e$  joining the configuration immediately after request  $r_{join}$  must be in state  $e[r_{join}]$  before it can participate in the running system.

An element is *self-stabilizing* [Dijkstra 1974] if its current state is completely defined by the previous  $k$  inputs it has processed for some fixed  $k$ . Running such an element long enough to ensure that it has

processed  $k$  inputs is all that is required to put it in state  $e[r_{join}]$ . Unfortunately, the design of self-stabilizing state machines is not always possible.

When elements are not self-stabilizing, processors are fail stop, and logical clocks are implemented, cooperation of a single state machine replica  $sm_i$  is sufficient to integrate a new element  $e$  into the system. This is because state information obtained from any state machine replica  $sm_i$  must be correct. In order to integrate  $e$  at request  $r_{join}$ , replica  $sm_i$  must have access to enough state information so that  $e[r_{join}]$  can be assembled and forwarded to  $e$ .

- When  $e$  is an output device,  $e[r_{join}]$  is likely to be only a small amount of device-specific setup information—information that changes infrequently and can be stored in state variables of  $sm_i$ .
- When  $e$  is a client, the information needed for  $e[r_{join}]$  is frequently based on recent sensor values read and can therefore be determined by using information provided to  $sm_i$  by other clients.
- And, when  $e$  is a state machine replica, the information needed for  $e[r_{join}]$  is stored in the state variables and pending requests at  $sm_i$ .

The protocol for integrating a client or output device  $e$  is simple— $e[r_{join}]$  is sent to  $e$  before the output produced by processing any request with a unique identifier larger than  $uid(r_{join})$ . The protocol for integrating a state machine replica  $sm_{new}$  is a bit more complex. It is not sufficient for replica  $sm_i$  simply to send the values of all its state variables and copies of any pending requests to  $sm_{new}$ . This is because some client request might be received by  $sm_i$  after sending  $e[r_{join}]$  but delivered to  $sm_{new}$  before its repair. Such a request would neither be reflected in the state information forwarded by  $sm_i$  to  $sm_{new}$  nor received by  $sm_{new}$  directly. Thus,  $sm_i$  must, for a time, relay to  $sm_{new}$  requests received from clients.<sup>10</sup> Since requests from a given client are received by  $sm_{new}$  in the order sent and in ascending order by request identifier,

<sup>10</sup> Duplicate copies of some requests might be received by  $sm_{new}$ .

once  $sm_{new}$  has received a request directly (i.e., not relayed) from a client  $c$ , there is no need for requests from  $c$  with larger identifiers to be relayed to  $sm_{new}$ . If  $sm_{new}$  informs  $sm_i$  of the identifier on a request received directly from each client  $c$ , then  $sm_i$  can know when to stop relaying to  $sm_{new}$  requests from  $c$ .

The complete integration protocol is summarized in the following:

**Integration with Fail-stop Processors and Logical Clocks.** A state machine replica  $sm_i$  can integrate an element  $e$  at request  $r_{join}$  into a running system as follows:

If  $e$  is a client or output device,  $sm_i$  sends the relevant portions of its state variables to  $e$  and does so before sending any output produced by requests with unique identifiers larger than the one on  $r_{join}$ .

If  $e$  is a state machine replica  $sm_{new}$ , then  $sm_i$

- (1) sends the values of its state variables and copies of any pending requests to  $sm_{new}$ ,

and then

- (2) sends to  $sm_{new}$  every subsequent request  $r$  received from each client  $c$  such that  $uid(r) < uid(r_c)$ , where  $r_c$  is the first request  $sm_{new}$  received directly from  $c$  after being restarted.

The existence of synchronized real-time clocks permits this protocol to be simplified because  $sm_i$  can determine when to stop relaying messages based on the passage of time. Suppose, as in Section 3.2.2, there exists a constant  $\Delta$  such that a request  $r$  with unique identifier  $uid(r)$  will be received by every (correct) state machine replica no later than time  $uid(r) + \Delta$  according to the local clock at the receiving processor. Let  $sm_{new}$  join the configuration at time  $\tau_{join}$ . By definition,  $sm_{new}$  is guaranteed to receive every request that was made after time  $\tau_{join}$  on the requesting client's clock. Since unique identifiers are obtained from the real-time clock of the client making the request,  $sm_{new}$  is guaranteed to receive every request  $r$  such that  $uid(r) \geq \tau_{join}$ . The first such request  $r$  must be received by  $sm_i$

by time  $\tau_{join} + \Delta$  according to its clock. Therefore, every request received by  $sm_i$  after  $\tau_{join} + \Delta$  must also be received directly by  $sm_{new}$ . Clearly,  $sm_i$  need not relay such requests, and we have the following protocol:

**Integration with Fail-stop Processors and Real-time Clocks.** A state machine replica  $sm_i$  can integrate an element  $e$  at request  $r_{join}$  into a running system as follows:

If  $e$  is a client or output device, then  $sm_i$  sends the relevant portions of its state variables to  $e$  and does so before sending any output produced by requests with unique identifiers larger than the one on  $r_{join}$ .

If  $e$  is a state machine replica  $sm_{new}$ , then  $sm_i$

- (1) sends the values of its state variables and copies of any pending requests to  $sm_{new}$ ,
- and then
- (2) sends to  $sm_{new}$  every request received during the next interval of duration  $\Delta$ .

When processors can exhibit Byzantine failures, a single state machine replica  $sm_i$  is not sufficient for integrating a new element into the system. This is because state information furnished by  $sm_i$  might not be correct— $sm_i$  might be executing on a faulty processor. To tolerate  $t$  failures in a system with  $2t + 1$  state machine replicas,  $t + 1$  identical copies of the state information and  $t + 1$  identical copies of relayed messages must be obtained. Otherwise, the protocol is as described above for real-time clocks.

### 7.2.1 Stability Revisited

The stability tests of Section 3 do not work when requests made by a client can be received from two sources—the client and via a relay. During the interval that messages are being relayed,  $sm_{new}$ , the state machine replica being integrated, might receive a request  $r$  directly from  $c$  but later receive  $r'$ , another request from  $c$ , with  $uid(r) > uid(r')$ , because  $r'$  was relayed by  $sm_i$ . The solution to this problem is for

$sm_{new}$  to consider requests received directly from  $c$  stable only after no relayed requests from  $c$  can arrive. Thus, the stability test must be changed:

**Stability Test During Restart.** A request  $r$  received directly from a client  $c$  by a restarting state machine replica  $sm_{new}$  is stable only after the last request from  $c$  relayed by another processor has been received by  $sm_{new}$ .

An obvious way to implement this new stability test is for a message to be sent to  $sm_{new}$  when no further requests from  $c$  will be relayed.

## 8. RELATED WORK

The state machine approach was first described in Lamport [1978a] for environments in which failures could not occur. It was generalized to handle fail-stop failures in Schneider [1982], a class of failures between fail-stop and Byzantine failures in Lamport [1978b], and full Byzantine failures in Lamport [1984]. These various state machine implementations were first characterized using the Agreement and Order requirements and a stability test in Schneider [1985].

The state machine approach has been used in the design of significant fault-tolerant process control applications [Wensley et al. 1978]. It has also been used in the design of distributed synchronization—including read/write locks and distributed semaphores [Schneider 1980], input/output guards for CSP and conditional Ada SELECT statements [Schneider 1982]—and in the design of a fail-stop processor approximation using processors that can exhibit arbitrary behavior in response to a failure [Schlichting and Schneider 1983; Schneider 1984]. A stable storage implementation described in Bernstein [1985] exploits properties of a synchronous broadcast network to avoid explicit protocols for Agreement and Order and uses Transmitting a Default Vote (as described in Section 7). The notion of  $\Delta$  common storage, suggested in Cristian et al. [1985], is a state machine implementation of memory that

uses the Real-time Clock Stability Test. The decentralized commit protocol of Skeen [1982] can be viewed as a straightforward application of the state machine approach, whereas the two-phase commit protocol described in Gray [1978] can be obtained from decentralized commit simply by making restrictive assumptions about failures and performing optimizations based on these assumptions. The Paxos Synod commit protocol [Lamport 1989] also can be understood in terms of the state machine approach. It is similar to, but less expensive to execute, than the standard three-phase commit protocol. Finally, the method of implementing highly available distributed services in Liskov and Ladin [1986] uses the state machine approach, with clever optimizations of the stability test and agreement protocol that are possible due to the semantics of the application and the use of fail-stop processors.

A critique of the state machine approach for transaction management in database systems appears in Garcia-Molina et al. [1986]. Experiments evaluating the performance of various of the stability tests in a network of SUN Workstations are reported in Pittelli and Garcia-Molina [1989]. That study also reports on the performance of request batching, which is possible when requests describe database transactions, and the use of null requests in the Logical Clock Stability Test Tolerating Fail-stop Failures of Section 3.

Primitives to support the Agreement and Order requirements for Replica Coordination have been included in two operating systems toolkits. The ISIS Toolkit [Birman 1985] provides ABCAST and CBCAST for allowing an applications programmer to control the delivery order of messages to the members of a process group (i.e., collection of state machine replicas). ABCAST ensures that all state machine replicas process requests in the same order; CBCAST allows more flexibility in message ordering and ensures that causally related requests are delivered in the correct relative order. ISIS has been used to implement a number of prototype applications. One example is the RNFS (replicated NFS) file system, a

network file system that is tolerant to fail-stop failures and runs on top of NFS, that was designed using the state machine approach [Marzullo and Schmuck 1988].

The Psync primitive [Peterson et al. 1989], which has been implemented in the *x*-kernel [Hutchinson and Peterson 1988], is similar to the CBCAST of ISIS. Psync, however, makes available to the programmer the graph of the message "potential causality" relation, whereas CBCAST does not. Psync is intended to be a low-level protocol that can be used to implement protocols like ABCAST and CBCAST; the ISIS primitives are intended for use by applications programmers and, therefore, hide the "potential causality" relation while at the same time include support for group management and failure reporting.

#### ACKNOWLEDGMENTS

This material is based on work supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grants Nos. DCR-8320274 and CCR-8701103, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

Discussions with O. Babaoglu, K. Birman, and L. Lamport over the past 5 years have helped me formulate the ideas in this paper. Useful comments on drafts of this paper were provided by J. Aizikowitz, O. Babaoglu, A. Bernstein, K. Birman, R. Brown, D. Gries, K. Marzullo, and B. Simons. I am very grateful to Sal March, managing editor of *ACM Computing Surveys*, for his thorough reading of this paper and many helpful comments.

#### REFERENCES

- AIZIKOWITZ, J. 1989. Designing distributed services using refinement mappings. Ph.D. dissertation, Computer Science Dept., Cornell Univ., Ithaca, New York. Also available as Tech. Rep. TR 89-1040.
- BERNSTEIN, A. J. 1985. A loosely coupled system for reliably storing data. *IEEE Trans. Softw. Eng.* SE-11, 5 (May), 446-454.
- BIRMAN, K. P. 1985. Replication and fault tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington, Dec. 1985), ACM, pp. 79-86.
- BIRMAN, K. P., AND JOSEPH, T. 1987. Reliable communication in the presence of failures. *ACM TOCS* 5, 1 (Feb. 1987), 47-76.
- CRISTIAN, F., AGHILI, H., STRONG, H. R., AND DOLEV, D. 1985. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Conference on Fault-tolerant Computing* (Ann Arbor, Mich., June 1985), IEEE Computer Society.
- DIJKSTRA, E. W. 1974. Self stabilization in spite of distributed control. *Commun. ACM* 17, 11 (Nov.), 643-644.
- FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (Apr. 1986), 374-382.
- GARCIA-MOLINA, H., PITTELLI, F., AND DAVIDSON, S. 1986. Application of Byzantine agreement in database systems. *ACM TODS* 11, 1 (Mar. 1986), 27-47.
- GOPAL, A., STRONG, R., TOUEG, S., AND CRISTIAN, F. 1990. Early-delivery atomic broadcast. To appear in *Proceedings of the 9th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Québec City, Québec, Aug. 1990).
- GRAY, J. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. Vol. 60. Springer-Verlag, New York, pp. 393-481.
- HALPERN, J., SIMONS, B., STRONG, R., AND DOLEV, D. 1984. Fault-tolerant clock synchronization. In *Proceedings of the 3rd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Vancouver, Canada, Aug.), pp. 89-102.
- HUTCHINSON, N., AND PETERSON, L. 1988. Design of the *x*-kernel. In *Proceedings of SIGCOMM '88—Symposium on Communication Architectures and Protocols* (Stanford, Calif., Aug.), pp. 65-75.
- LAMPORT, L. 1978a. Time, clocks and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July), 558-565.
- LAMPORT, L. 1979b. The implementation of reliable distributed multiprocess systems. *Comput. Networks* 2, 95-114.
- LAMPORT, L. 1984. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS* 6, 2 (Apr.), 254-280.
- LAMPORT, L. 1989. The part-time parliament. Tech. Rep. 49. Digital Equipment Corporation Systems Research Center, Palo Alto, Calif.
- LAMPORT, L., AND MELLAR-SMITH, P. M. 1984. Byzantine clock synchronization. In *Proceedings of the 3rd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Vancouver, Canada, Aug.), 68-74.
- LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM TOPLAS* 4, 3 (July), 382-401.

- LISKOV, B., AND LADIN, R. 1986. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada, Aug.), ACM, pp. 29–39.
- MANCINI, L., AND PAPPALARDO, G. 1988. Towards a theory of replicated processing. *Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science*, Vol. 331. Springer-Verlag, New York, pp. 175–192.
- MARZULLO, K. 1989. Implementing fault-tolerant sensors. Tech. Rep. TR 89-997. Computer Science Dept., Cornell Univ., Ithaca, New York.
- MARZULLO, K., AND SCHMUCK, F. 1988. Supplying high availability with a standard network file system. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (San Jose, CA, June), IEEE Computer Society, pp. 447–455.
- PETERSON, L. L., BUCHOLZ, N. C., AND SCHLICHTING, R. D. 1989. Preserving and using context information in interprocess communication. *ACM TOCS* 7, 3 (Aug.), 217–246.
- PITTELLI, F. M., AND GARCIA-MOLINA, H. 1989. Reliable scheduling in a TMR database system. *ACM TOCS* 7, 1 (Feb.), 25–60.
- SCHLICHTING, R. D., AND SCHNEIDER, F. B. 1983. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *ACM TOCS* 1, 3 (Aug.), 222–238.
- SCHNEIDER, F. B. 1980. Ensuring consistency on a distributed database system by use of distributed semaphores. In *Proceedings of International Symposium on Distributed Data Bases* (Paris, France, Mar.), INRIA, pp. 183–189.
- SCHNEIDER, F. B. 1982. Synchronization in distributed programs. *ACM TOPLAS* 4, 2 (Apr.), 179–195.
- SCHNEIDER, F. B. 1984. Byzantine generals in action: Implementing fail-stop processors. *ACM TOCS* 2, 2 (May), 145–154.
- SCHNEIDER, F. B. 1985. Paradigms for distributed programs. *Distributed Systems. Methods and Tools for Specification. Lecture Notes in Computer Science*, Vol. 190. Springer-Verlag, New York, pp. 343–430.
- SCHNEIDER, F. B. 1986. A paradigm for reliable clock synchronization. In *Proceedings of the Advanced Seminar on Real-Time Local Area Networks* (Bandol, France, Apr.), INRIA, pp. 85–104.
- SCHNEIDER, F. B., GRIES, D., AND SCHLICHTING, R. D. 1984. Fault-tolerant broadcasts. *Sci. Comput. Program.* 4, 1–15.
- SIEWIOREK, D. P., AND SWARZ, R. S. 1982. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass.
- SKEEN, D. 1982. Crash recovery in a distributed database system. Ph.D. dissertation, Univ. of California at Berkeley, May.
- STRONG, H. R., AND DOLEV, D. 1983. Byzantine agreement. *Intellectual Leverage for the Information Society, Digest of Papers*. (Compcon 83, IEEE Computer Society, Mar.), IEEE Computer Society, pp. 77–82.
- WENSLEY, J., WENSKY, J. H., LAMPORT, L., GOLDBERG, J., GREEN, M. W., LEVITT, K. N., MELLAR-SMITH, P. M., SHOSTAK, R. E., and WEINSTOCK, C. B. 1978. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE* 66, 10 (Oct.), 1240–1255.

Received November 1987; final revision accepted January 1990.



## Chapter 10

# Data Processing - External Sorting

This chapter contains the book chapter:

H. Garcia-Molina, J. D. Ullman, J. Widom. Database Systems: The Complete Book. Chapter 11.4, pp. 525–533 (9 of 1119). Prentice Hall, 2002. ISBN: 0-13-031995-3

A vast number of applications must process and analyze large quantities of data. These applications need the system property of **scalability** with data volumes. Classic algorithms are developed with the RAM model in mind, and often break or perform very poorly over multi-level memories. This necessitates the study of algorithms under different models. In the following, the text studies algorithms for data processing under the external memory model. In this chapter, the ubiquitous problem of *sorting* is studied. *The ultimate goal of this portion of the material is to allow us to reflect on how the external memory model differs from the classic model of algorithm design, and to explore in detail the example of sorting as a first data processing service implemented under the external memory model.*

The learning goals for this portion of the material are listed below.

- Identify the problem of processing large data collections with external memory algorithms.
- Discuss the implications of the external memory model on algorithm design.
- Explain the classic sort-merge external sorting algorithm, and in particular its two-phase, multiway variants.

*This page has intentionally been left blank.*



**!! Exercise 11.3.4:** At the end of Example 11.3 we suggested that the maximum density of tracks could be reduced if we divided the tracks into three regions, with different numbers of sectors in each region. If the divisions between the three regions could be placed at any radius, and the number of sectors in each region could vary, subject only to the constraint that the total number of bytes on the 16,384 tracks of one surface be 8 gigabytes, what choice for the five parameters (radii of the two divisions between regions and the numbers of sectors per track in each of the three regions) minimizes the maximum density of any track?

## 11.4 Using Secondary Storage Effectively

In most studies of algorithms, one assumes that the data is in main memory, and access to any item of data takes as much time as any other. This model of computation is often called the “RAM model” or random-access model of computation. However, when implementing a DBMS, one must assume that the data does *not* fit into main memory. One must therefore take into account the use of secondary, and perhaps even tertiary storage in designing efficient algorithms. The best algorithms for processing very large amounts of data thus often differ from the best main-memory algorithms for the same problem.

In this section, we shall consider primarily the interaction between main and secondary memory. In particular, there is a great advantage in choosing an algorithm that uses few disk accesses, even if the algorithm is not very efficient when viewed as a main-memory algorithm. A similar principle applies at each level of the memory hierarchy. Even a main-memory algorithm can sometimes be improved if we remember the size of the cache and design our algorithm so that data moved to cache tends to be used many times. Likewise, an algorithm using tertiary storage needs to take into account the volume of data moved between tertiary and secondary memory, and it is wise to minimize this quantity even at the expense of more work at the lower levels of the hierarchy.

### 11.4.1 The I/O Model of Computation

Let us imagine a simple computer running a DBMS and trying to serve a number of users who are accessing the database in various ways: queries and database modifications. For the moment, assume our computer has one processor, one disk controller, and one disk. The database itself is much too large to fit in main memory. Key parts of the database may be buffered in main memory, but generally, each piece of the database that one of the users accesses will have to be retrieved initially from disk.

Since there are many users, and each user issues disk-I/O requests frequently, the disk controller often will have a queue of requests, which we assume it satisfies on a first-come-first-served basis. Thus, each request for a given user will appear random (i.e., the disk head will be in a random position before the

request), even if this user is reading blocks belonging to a single relation, and that relation is stored on a single cylinder of the disk. Later in this section we shall discuss how to improve the performance of the system in various ways. However, in all that follows, the following rule, which defines the *I/O model of computation*, is assumed:

**Dominance of I/O cost:** If a block needs to be moved between disk and main memory, then the time taken to perform the read or write is much larger than the time likely to be used manipulating that data in main memory. Thus, the number of block accesses (reads and writes) is a good approximation to the time needed by the algorithm and should be minimized.

In examples, we shall assume that the disk is a Megatron 747, with 16K-byte blocks and the timing characteristics determined in Example 11.5. In particular, the average time to read or write a block is about 11 milliseconds.

**Example 11.6:** Suppose our database has a relation  $R$  and a query asks for the tuple of  $R$  that has a certain key value  $k$ . As we shall see, it is quite desirable that an index on  $R$  be created and used to identify the disk block on which the tuple with key value  $k$  appears. However it is generally unimportant whether the index tells us where on the block this tuple appears.

The reason is that it will take on the order of 11 milliseconds to read this 16K-byte block. In 11 milliseconds, a modern microprocessor can execute millions of instructions. However, searching for the key value  $k$  once the block is in main memory will only take thousands of instructions, even if the dumbest possible linear search is used. The additional time to perform the search in main memory will therefore be less than 1% of the block access time and can be neglected safely.  $\square$

#### 11.4.2 Sorting Data in Secondary Storage

As an extended example of how algorithms need to change under the I/O model of computation cost, let us consider sorting data that is much larger than main memory. To begin, we shall introduce a particular sorting problem and give some details of the machine on which the sorting occurs.

**Example 11.7:** Let us assume that we have a large relation  $R$  consisting of 10,000,000 tuples. Each tuple is represented by a record with several fields, one of which is the *sort key* field, or just “key field” if there is no confusion with other kinds of keys. The goal of a sorting algorithm is to order the records by increasing value of their sort keys.

A sort key may or may not be a “key” in the usual SQL sense of a *primary key*, where records are guaranteed to have unique values in their primary key. If duplicate values of the sort key are permitted, then any order of records with equal sort keys is acceptable. For simplicity, we shall assume sort keys are unique.

The records (tuples) of  $R$  will be divided into disk blocks of 16,384 bytes per block. We assume that 100 records fit in one block. That is, records are about 160 bytes long. With the typical extra information needed to store records in a block (as discussed in Section 12.2, e.g.), 100 records of this size is about what can fit in one 16,384-byte block. Thus,  $R$  occupies 100,000 blocks totaling 1.64 billion bytes.

The machine on which the sorting occurs has one Megatron 747 disk and 100 megabytes of main memory available for buffering blocks of the relation. The actual main memory is larger, but the rest of main-memory is used by the system. The number of blocks that can fit in 100M bytes of memory (which, recall, is really  $100 \times 2^{20}$  bytes), is  $100 \times 2^{20}/2^{14}$ , or 6400 blocks.  $\square$

If the data fits in main memory, there are a number of well-known algorithms that work well;<sup>5</sup> variants of “Quicksort” are generally considered the fastest. The preferred version of Quicksort sorts only the key fields, carrying pointers to the full records along with the keys. Only when the keys and their pointers were in sorted order, would we use the pointers to bring every record to its proper position.

Unfortunately, these ideas do not work very well when secondary memory is needed to hold the data. The preferred approaches to sorting, when the data is mostly in secondary memory, involve moving each block between main and secondary memory only a small number of times, in a regular pattern. Often, these algorithms operate in a small number of *passes*; in one pass every record is read into main memory once and written out to disk once. In Section 11.4.4, we see one such algorithm.

### 11.4.3 Merge-Sort

You may be familiar with a main-memory sorting algorithm called Merge-Sort that works by merging sorted lists into larger sorted lists. To *merge* two sorted lists, we repeatedly compare the smallest remaining keys of each list, move the record with the smaller key to the output, and repeat, until one list is exhausted. At that time, the output, in the order selected, followed by what remains of the nonexhausted list, is the complete set of records, in sorted order.

**Example 11.8:** Suppose we have two sorted lists of four records each. To make matters simpler, we shall represent records by their keys and no other data, and we assume keys are integers. One of the sorted lists is  $(1, 3, 4, 9)$  and the other is  $(2, 5, 7, 8)$ . In Fig. 11.10 we see the stages of the merge process.

At the first step, the head elements of the two lists, 1 and 2, are compared. Since  $1 < 2$ , the 1 is removed from the first list and becomes the first element of the output. At step (2), the heads of the remaining lists, now 3 and 2, are compared; 2 wins and is moved to the output. The merge continues until

---

<sup>5</sup>See D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd Edition, Addison-Wesley, Reading MA, 1998.

Step	List 1	List 2	Output
start	1, 3, 4, 9	2, 5, 7, 8	none
1)	3, 4, 9	2, 5, 7, 8	1
2)	3, 4, 9	5, 7, 8	1, 2
3)	4, 9	5, 7, 8	1, 2, 3
4)	9	5, 7, 8	1, 2, 3, 4
5)	9	7, 8	1, 2, 3, 4, 5
6)	9	8	1, 2, 3, 4, 5, 7
7)	9	none	1, 2, 3, 4, 5, 7, 8
8)	none	none	1, 2, 3, 4, 5, 7, 8, 9

Figure 11.10: Merging two sorted lists to make one sorted list

step (7), when the second list is exhausted. At that point, the remainder of the first list, which happens to be only one element, is appended to the output and the merge is done. Note that the output is in sorted order, as must be the case, because at each step we chose the smallest of the remaining elements.  $\square$

The time to merge in main memory is linear in the sum of the lengths of the lists. The reason is that, because the given lists are sorted, only the heads of the two lists are ever candidates for being the smallest unselected element, and we can compare them in a constant amount of time. The classic merge-sort algorithm sorts recursively, using  $\log_2 n$  phases if there are  $n$  elements to be sorted. It can be described as follows:

**BASIS:** If there is a list of one element to be sorted, do nothing, because the list is already sorted.

**INDUCTION:** If there is a list of more than one element to be sorted, then divide the list arbitrarily into two lists that are either of the same length, or as close as possible if the original list is of odd length. Recursively sort the two sublists. Then merge the resulting sorted lists into one sorted list.

The analysis of this algorithm is well known and not too important here. Briefly  $T(n)$ , the time to sort  $n$  elements, is some constant times  $n$  (to split the list and merge the resulting sorted lists) plus the time to sort two lists of size  $n/2$ . That is,  $T(n) = 2T(n/2) + an$  for some constant  $a$ . The solution to this recurrence equation is  $T(n) = O(n \log n)$ , that is, proportional to  $n \log n$ .

#### 11.4.4 Two-Phase, Multiway Merge-Sort

We shall use a variant of Merge-Sort, called *Two-Phase, Multiway Merge-Sort* (often abbreviated TPMMS), to sort the relation of Example 11.7 on the machine described in that example. It is the preferred sorting algorithm in many database applications. Briefly, this algorithm consists of:

- *Phase 1:* Sort main-memory-sized pieces of the data, so every record is part of a sorted list that just fits in the available main memory. There may thus be any number of these *sorted sublists*, which we merge in the next phase.
- *Phase 2:* Merge all the sorted sublists into a single sorted list.

Our first observation is that with data on secondary storage, we do not want to start with a basis to the recursion that is one record or a few records. The reason is that Merge-Sort is not as fast as some other algorithms when the records to be sorted fit in main memory. Thus, we shall begin the recursion by taking an entire main memory full of records, and sorting them using an appropriate main-memory sorting algorithm such as Quicksort. We repeat the following process as many times as necessary:

1. Fill all available main memory with blocks from the original relation to be sorted.
2. Sort the records that are in main memory.
3. Write the sorted records from main memory onto new blocks of secondary memory, forming one sorted sublist.

At the end of this *first phase*, all the records of the original relation will have been read once into main memory, and become part of a main-memory-size *sorted sublist* that has been written onto disk.

**Example 11.9:** Consider the relation described in Example 11.7. We determined that 6400 of the 100,000 blocks will fill main memory. We thus fill memory 16 times, sort the records in main memory, and write the sorted sublists out to disk. The last of the 16 sublists is shorter than the rest; it occupies only 4000 blocks, while the other 15 sublists occupy 6400 blocks.

How long does this phase take? We read each of the 100,000 blocks once, and we write 100,000 new blocks. Thus, there are 200,000 disk I/O's. We have assumed, for the moment, that blocks are stored at random on the disk, an assumption that, as we shall see in Section 11.5, can be improved upon greatly. However, on our randomness assumption, each block read or write takes about 11 milliseconds. Thus, the I/O time for the first phase is 2200 seconds, or 37 minutes, or over 2 minutes per sorted sublist. It is not hard to see that, at a processor speed of hundreds of millions of instructions per second, we can create one sorted sublist in main memory in far less than the I/O time for that sublist. We thus estimate the total time for phase one as 37 minutes.  $\square$

Now, let us consider how we complete the sort by merging the sorted sublists. We could merge them in pairs, as in the classical Merge-Sort, but that would involve reading all data in and out of memory  $2 \log_2 n$  times if there were  $n$  sorted sublists. For instance, the 16 sorted sublists of Example 11.9 would be

read in and out of secondary storage once to merge into 8 lists; another complete reading and writing would reduce them to 4 sorted lists, and two more complete read/write operations would reduce them to one sorted list. Thus, each block would have 8 disk I/O's performed on it.

A better approach is to read the first block of each sorted sublist into a main-memory buffer. For some huge relations, there would be too many sorted sublists from phase one to read even one block per list into main memory, a problem we shall deal with in Section 11.4.5. But for data such as that of Example 11.7, there are relatively few lists, 16 in that example, and a block from each list fits easily in main memory.

We also use a buffer for an output block that will contain as many of the first elements in the complete sorted list as it can hold. Initially, the output block is empty. The arrangement of buffers is suggested by Fig. 11.11. We merge the sorted sublists into one sorted list with all the records as follows.

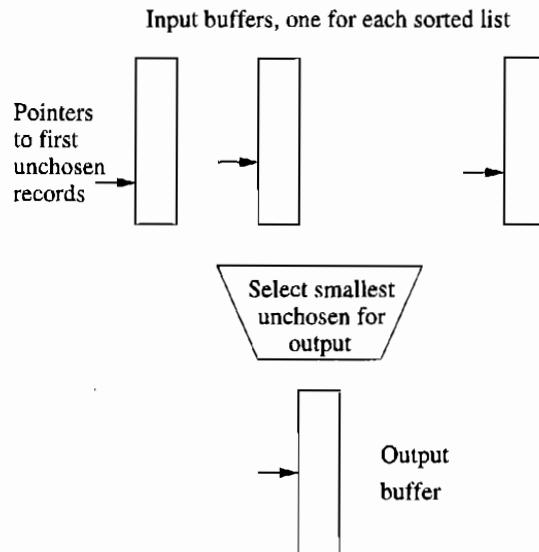


Figure 11.11: Main-memory organization for multiway merging

1. Find the smallest key among the first remaining elements of all the lists. Since this comparison is done in main memory, a linear search is sufficient, taking a number of machine instructions proportional to the number of sublists. However, if we wish, there is a method based on “priority queues”<sup>6</sup> that takes time proportional to the logarithm of the number of sublists to find the smallest element.
2. Move the smallest element to the first available position of the output block.

---

<sup>6</sup>See Aho, A. V. and J. D. Ullman *Foundations of Computer Science*, Computer Science Press, 1992.

### How Big Should Blocks Be?

We have assumed a 16K byte block in our analysis of algorithms using the Megatron 747 disk. However, there are arguments that a larger block size would be advantageous. Recall from Example 11.5 that it takes about a quarter of a millisecond for transfer time of a 16K block and 10.63 milliseconds for average seek time and rotational latency. If we doubled the size of blocks, we would halve the number of disk I/O's for an algorithm like TPMMS. On the other hand, the only change in the time to access a block would be that the transfer time increases to 0.50 millisecond. We would thus approximately halve the time the sort takes. For a block size of 512K (i.e., an entire track of the Megatron 747) the transfer time is 8 milliseconds. At that point, the average block access time would be 20 milliseconds, but we would need only 12,500 block accesses, for a speedup in sorting by a factor of 14.

However, there are reasons to limit the block size. First, we cannot use blocks that cover several tracks effectively. Second, small relations would occupy only a fraction of a block, so large blocks would waste space on the disk. There are also certain data structures for secondary storage organization that prefer to divide data among many blocks and therefore work less well when the block size is too large. In fact, we shall see in Section 11.4.5 that the larger the blocks are, the fewer records we can sort by TPMMS. Nevertheless, as machines get faster and disks more capacious, there is a tendency for block sizes to grow.

3. If the output block is full, write it to disk and reinitialize the same buffer in main memory to hold the next output block.
4. If the block from which the smallest element was just taken is now exhausted of records, read the next block from the same sorted sublist into the same buffer that was used for the block just exhausted. If no blocks remain, then leave its buffer empty and do not consider elements from that list in any further competition for smallest remaining elements.

In the second phase, unlike the first phase, the blocks are read in an unpredictable order, since we cannot tell when an input block will become exhausted. However, notice that every block holding records from one of the sorted lists is read from disk exactly once. Thus, the total number of block reads is 100,000 in the second phase, just as for the first. Likewise, each record is placed once in an output block, and each of these blocks is written to disk. Thus, the number of block writes in the second phase is also 100,000. As the amount of second-phase computation in main memory can again be neglected compared to the I/O cost, we conclude that the second phase takes another 37 minutes, or 74

minutes for the entire sort.

#### 11.4.5 Multiway Merging of Larger Relations

The Two-Phase, Multiway Merge-Sort (TPMMS) described above can be used to sort some very large sets of records. To see how large, let us suppose that:

1. The block size is  $B$  bytes.
2. The main memory available for buffering blocks is  $M$  bytes.
3. Records take  $R$  bytes.

The number of buffers available in main memory is thus  $M/B$ . On the second phase, all but one of these buffers may be devoted to one of the sorted sublists; the remaining buffer is for the output block. Thus, the number of sorted sublists that may be created in phase one is  $(M/B) - 1$ . This quantity is also the number of times we may fill main memory with records to be sorted. Each time we fill main memory, we sort  $M/R$  records. Thus, the total number of records we can sort is  $(M/R)((M/B)-1)$ , or approximately  $M^2/RB$  records.

**Example 11.10:** If we use the parameters outlined in Example 11.7, then  $M = 104,857,600$ ,  $B = 16,384$ , and  $R = 160$ . We can thus sort up to  $M^2/RB = 4.2$  billion records, occupying two thirds of a terabyte. Note that relations this size will not fit on a Megatron 747 disk.  $\square$

If we need to sort more records, we can add a third pass. Use TPMMS to sort groups of  $M^2/RB$  records, turning them into sorted sublists. Then, in a third phase, we merge up to  $(M/B) - 1$  of these lists in a final multiway merge.

The third phase lets us sort approximately  $M^3/RB^2$  records occupying  $M^3/B^3$  blocks. For the parameters of Example 11.7, this amount is about 27 trillion records occupying 4.3 petabytes. Such an amount is unheard of today. Since even the 0.67 terabyte limit for TPMMS is unlikely to be carried out in secondary storage, we suggest that the two-phase version of Multiway Merge-Sort is likely to be enough for all practical purposes.

#### 11.4.6 Exercises for Section 11.4

**Exercise 11.4.1:** Using TPMMS, how long would it take to sort the relation of Example 11.7 if the Megatron 747 disk were replaced by the Megatron 777 disk described in Exercise 11.3.1, and all other characteristics of the machine and data remained the same?

**Exercise 11.4.2:** Suppose we use TPMMS on the machine and relation  $R$  of Example 11.7, with certain modifications. Tell how many disk I/O's are needed for the sort if the relation  $R$  and/or machine characteristics are changed as follows:

- \* a) The number of tuples in  $R$  is doubled (all else remains the same).
- b) The length of tuples is doubled to 320 bytes (and everything else remains as in Example 11.7).
- \* c) The size of blocks is doubled, to 32,768 bytes (again, as throughout this exercise, all other parameters are unchanged).
- d) The size of available main memory is doubled to 200 megabytes.

**! Exercise 11.4.3:** Suppose the relation  $R$  of Example 11.7 grows to have as many tuples as can be sorted using TPMMS on the machine described in that example. Also assume that the disk grows to accommodate  $R$ , but all other characteristics of the disk, machine, and relation  $R$  remain the same. How long would it take to sort  $R$ ?

**\* Exercise 11.4.4:** Let us again consider the relation  $R$  of Example 11.7, but assume that it is stored sorted by the sort key (which is in fact a “key” in the usual sense, and uniquely identifies records). Also, assume that  $R$  is stored in a sequence of blocks whose locations are known, so that for any  $i$  it is possible to locate and retrieve the  $i$ th block of  $R$  using one disk I/O. Given a key value  $K$ , we can find the tuple with that key value by using a standard binary search technique. What is the maximum number of disk I/O’s needed to find the tuple with key  $K$ ?

**!! Exercise 11.4.5:** Suppose we have the same situation as in Exercise 11.4.4, but we are given 10 key values to find. What is the maximum number of disk I/O’s needed to find all 10 tuples?

**\* Exercise 11.4.6:** Suppose we have a relation whose  $n$  tuples each require  $R$  bytes, and we have a machine whose main memory  $M$  and disk-block size  $B$  are just sufficient to sort the  $n$  tuples using TPMMS. How would the maximum  $n$  change if we doubled: (a)  $B$  (b)  $R$  (c)  $M$ ?

**! Exercise 11.4.7:** Repeat Exercise 11.4.6 if it is just possible to perform the sort using Three-Phase, Multiway Merge-Sort.

**\*! Exercise 11.4.8:** As a function of parameters  $R$ ,  $M$ , and  $B$  (as in Exercise 11.4.6) and the integer  $k$ , how many records can be sorted using a  $k$ -phase, Multiway Merge-Sort?

## 11.5 Accelerating Access to Secondary Storage

The analysis of Section 11.4.4 assumed that data was stored on a single disk and that blocks were chosen randomly from the possible locations on the disk. That assumption may be appropriate for a system that is executing a large number of small queries simultaneously. But if all the system is doing is sorting a large



## Chapter 11

# Data Processing - Basic Relational Operators and Joins

This chapter contains the book chapter:

H. Garcia-Molina, J. D. Ullman, J. Widom. Database Systems: The Complete Book. Chapter 15, pp. 713–774 (62 of 1119). Prentice Hall, 2002. ISBN: 0-13-031995-3

Other than sorting, a set of operations commonly applied to bulk data sets are the operations of the relational algebra. This chapter discusses how to design algorithms for and implement these operators under the external memory model, maintaining the property of **scalability** with data volumes. *The ultimate goal of this portion of the material is to equip us with a variety of algorithmic design alternatives for services which must analyze large data sets, in particular memory management approaches, indexing, sorting, and hashing.*

The learning goals for this portion of the material are listed below.

- Explain the design of operators for basic relational operations, including their scan-based, one-pass variants as well as associated memory management issues.
- Analyze algorithms for the join operation, including the multiple variants based on nested loops, indexing, sorting, and hashing as well as one, two, and many passes.
- Apply external memory algorithms to the implementation of data processing operators.

*This page has intentionally been left blank.*

# Chapter 15

# Query Execution

Previous chapters gave us data structures that allow efficient execution of basic database operations such as finding tuples given a search key. We are now ready to use these structures to support efficient algorithms for answering queries. The broad topic of query processing will be covered in this chapter and Chapter 16. The *query processor* is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes those operations. Since SQL lets us express queries at a very high level, the query processor must supply a lot of detail regarding how the query is to be executed. Moreover, a naive execution strategy for a query may lead to an algorithm for executing the query that takes far more time than necessary.

Figure 15.1 suggests the division of topics between Chapters 15 and 16. In this chapter, we concentrate on query execution, that is, the algorithms that manipulate the data of the database. We focus on the operations of the extended relational algebra, described in Section 5.4. Because SQL uses a bag model, we also assume that relations are bags, and thus use the bag versions of the operators from Section 5.3.

We shall cover the principal methods for execution of the operations of relational algebra. These methods differ in their basic strategy; scanning, hashing, sorting, and indexing are the major approaches. The methods also differ on their assumption as to the amount of available main memory. Some algorithms assume that enough main memory is available to hold at least one of the relations involved in an operation. Others assume that the arguments of the operation are too big to fit in memory, and these algorithms have significantly different costs and structures.

## Preview of Query Compilation

Query compilation is divided into the three major steps shown in Fig. 15.2.

- a) *Parsing*, in which a *parse tree*, representing the query and its structure, is constructed.

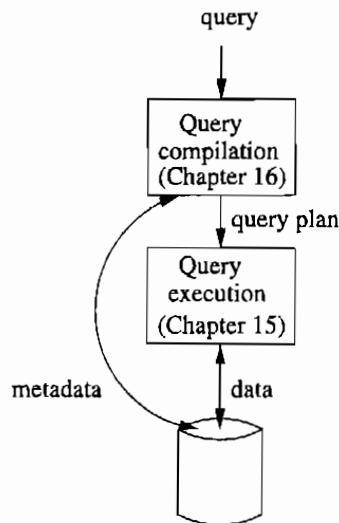


Figure 15.1: The major parts of the query processor

- b) *Query rewrite*, in which the parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.
- c) *Physical plan generation*, where the abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. Chapter 16 is devoted to query optimization; we shall learn there how to select a “query plan” that takes as little time as possible. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?
2. For each operation of the selected form, what algorithm should we use to implement that operation?
3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each

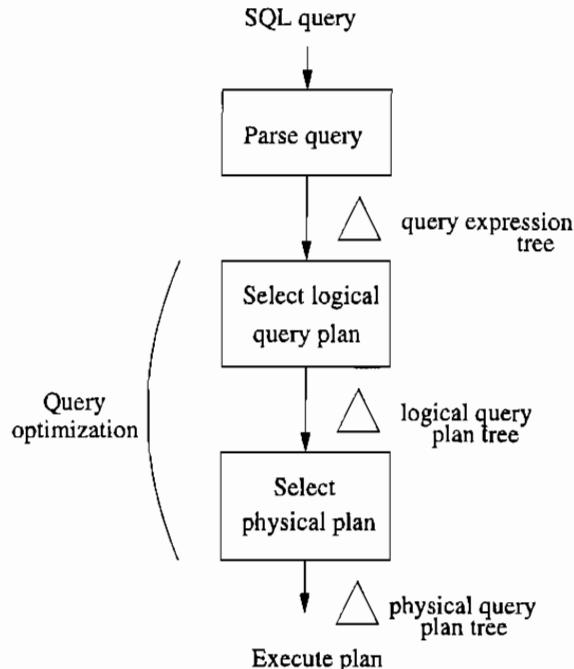


Figure 15.2: Outline of query compilation

relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

## 15.1 Introduction to Physical-Query-Plan Operators

Physical query plans are built from operators, each of which implements one step of the plan. Often, the physical operators are particular implementations for one of the operators of relational algebra. However, we also need physical operators for other tasks that do not involve an operator of relational algebra. For example, we often need to “scan” a table, that is, bring into main memory each tuple of some relation that is an operand of a relational-algebra expression. In this section, we shall introduce the basic building blocks of physical query plans. Later sections cover the more complex algorithms that implement operators of relational algebra efficiently; these algorithms also form an essential part of physical query plans. We also introduce here the “iterator” concept, which is an important method by which the operators comprising a physical query plan can pass requests for tuples and answers among themselves.

### 15.1.1 Scanning Tables

Perhaps the most basic thing we can do in a physical query plan is to read the entire contents of a relation  $R$ . This step is necessary when, for example, we take the union or join of  $R$  with another relation. A variation of this operator involves a simple predicate, where we read only those tuples of the relation  $R$  that satisfy the predicate. There are two basic approaches to locating the tuples of a relation  $R$ .

1. In many cases, the relation  $R$  is stored in an area of secondary memory, with its tuples arranged in blocks. The blocks containing the tuples of  $R$  are known to the system, and it is possible to get the blocks one by one. This operation is called *table-scan*.
2. If there is an index on any attribute of  $R$ , we may be able to use this index to get all the tuples of  $R$ . For example, a sparse index on  $R$ , as discussed in Section 13.1.3, can be used to lead us to all the blocks holding  $R$ , even if we don't know otherwise which blocks these are. This operation is called *index-scan*.

We shall take up index-scan again in Section 15.6.2, when we talk about implementation of the  $\sigma$  operator. However, the important observation for now is that we can use the index not only to get *all* the tuples of the relation it indexes, but to get only those tuples that have a particular value (or sometimes a particular range of values) in the attribute or attributes that form the search key for the index.

### 15.1.2 Sorting While Scanning Tables

There are a number of reasons why we might want to sort a relation as we read its tuples. For one, the query could include an ORDER BY clause, requiring that a relation be sorted. For another, various algorithms for relational-algebra operations require one or both of their arguments to be sorted relations. These algorithms appear in Section 15.4 and elsewhere.

The physical-query-plan operator *sort-scan* takes a relation  $R$  and a specification of the attributes on which the sort is to be made, and produces  $R$  in that sorted order. There are several ways that sort-scan can be implemented:

- a) If we are to produce a relation  $R$  sorted by attribute  $a$ , and there is a B-tree index on  $a$ , or  $R$  is stored as an indexed-sequential file ordered by  $a$ , then a scan of the index allows us to produce  $R$  in the desired order.
- b) If the relation  $R$  that we wish to retrieve in sorted order is small enough to fit in main memory, then we can retrieve its tuples using a table scan or index scan, and then use one of many possible efficient, main-memory sorting algorithms.

- c) If  $R$  is too large to fit in main memory, then the multiway merging approach covered in Section 11.4.3 is a good choice. However, instead of storing the final sorted  $R$  back on disk, we produce one block of the sorted  $R$  at a time, as its tuples are needed.

### 15.1.3 The Model of Computation for Physical Operators

A query generally consists of several operations of relational algebra, and the corresponding physical query plan is composed of several physical operators. Often, a physical operator is an implementation of a relational-algebra operator, but as we saw in Section 15.1.1, other physical plan operators correspond to operations like scanning that may be invisible in relational algebra.

Since choosing physical plan operators wisely is an essential of a good query processor, we must be able to estimate the “cost” of each operator we use. We shall use the number of disk I/O’s as our measure of cost for an operation. This measure is consistent with our view (see Section 11.4.1) that it takes longer to get data from disk than to do anything useful with it once the data is in main memory. The one major exception is when answering a query involves communicating data across a network. We discuss costs for distributed query processing in Sections 15.9 and 19.4.4.

When comparing algorithms for the same operations, we shall make an assumption that may be surprising at first:

- We assume that the arguments of any operator are found on disk, but the result of the operator is left in main memory.

If the operator produces the final answer to a query, and that result is indeed written to disk, then the cost of doing so depends only on the size of the answer, and not on how the answer was computed. We can simply add the final write-back cost to the total cost of the query. However, in many applications, the answer is not stored on disk at all, but printed or passed to some formatting program. Then, the disk I/O cost of the output either is zero or depends upon what some unknown application program does with the data.

Similarly, the result of an operator that forms part of a query (rather than the whole query) often is not written to disk. In Section 15.1.6 we shall discuss “iterators,” where the result of one operator is constructed in main memory, perhaps a small piece at a time, and passed as an argument to another operator. In this situation, we never have to write the result to disk, and moreover, we save the cost of reading from disk this argument of the operator that uses the result. This saving is an excellent opportunity for the query optimizer.

### 15.1.4 Parameters for Measuring Costs

Now, let us introduce the parameters (sometimes called statistics) that we use to express the cost of an operator. Estimates of cost are essential if the optimizer

is to determine which of the many query plans is likely to execute fastest. Section 16.5 introduces the exploitation of these cost estimates.

We need a parameter to represent the portion of main memory that the operator uses, and we require other parameters to measure the size of its argument(s). Assume that main memory is divided into buffers, whose size is the same as the size of disk blocks. Then  $M$  will denote the number of main-memory buffers available to an execution of a particular operator. When evaluating the cost of an operator, we shall not count the cost — either memory used or disk I/O's — of producing the output; thus  $M$  includes only the space used to hold the input and any intermediate results of the operator.

Sometimes, we can think of  $M$  as the entire main memory, or most of the main memory, as we did in Section 11.4.4. However, we shall also see situations where several operations share the main memory, so  $M$  could be much smaller than the total main memory. In fact, as we shall discuss in Section 15.7, the number of buffers available to an operation may not be a predictable constant, but may be decided during execution, based on what other processes are executing at the same time. If so,  $M$  is really an estimate of the number of buffers available to the operation. If the estimate is wrong, then the actual execution time will differ from the predicted time used by the optimizer. We could even find that the chosen physical query plan would have been different, had the query optimizer known what the true buffer availability would be during execution.

Next, let us consider the parameters that measure the cost of accessing argument relations. These parameters, measuring size and distribution of data in a relation, are often computed periodically to help the query optimizer choose physical operators.

We shall make the simplifying assumption that data is accessed one block at a time from disk. In practice, one of the techniques discussed in Section 11.5 might be able to speed up the algorithm if we are able to read many blocks of the relation at once, and they can be read from consecutive blocks on a track. There are three parameter families,  $B$ ,  $T$ , and  $V$ :

- When describing the size of a relation  $R$ , we most often are concerned with the number of blocks that are needed to hold all the tuples of  $R$ . This number of blocks will be denoted  $B(R)$ , or just  $B$  if we know that relation  $R$  is meant. Usually, we assume that  $R$  is *clustered*; that is, it is stored in  $B$  blocks or in approximately  $B$  blocks. As discussed in Section 13.1.6, we may in fact wish to keep a small fraction of each block holding  $R$  empty for future insertions into  $R$ . Nevertheless,  $B$  will often be a good-enough approximation to the number of blocks that we must read from disk to see all of  $R$ , and we shall use  $B$  as that estimate uniformly.
- Sometimes, we also need to know the number of tuples in  $R$ , and we denote this quantity by  $T(R)$ , or just  $T$  if  $R$  is understood. If we need the number of tuples of  $R$  that can fit in one block, we can use the ratio  $T/B$ . Further, there are some instances where a relation is stored distributed

among blocks that are also occupied by tuples of other relations. If so, then a simplifying assumption is that each tuple of  $R$  requires a separate disk read, and we shall use  $T$  as an estimate of the disk I/O's needed to read  $R$  in this situation.

- Finally, we shall sometimes want to refer to the number of distinct values that appear in a column of a relation. If  $R$  is a relation, and one of its attributes is  $a$ , then  $V(R, a)$  is the number of distinct values of the column for  $a$  in  $R$ . More generally, if  $[a_1, a_2, \dots, a_n]$  is a list of attributes, then  $V(R, [a_1, a_2, \dots, a_n])$  is the number of distinct  $n$ -tuples in the columns of  $R$  for attributes  $a_1, a_2, \dots, a_n$ . Put formally, it is the number of tuples in  $\delta(\pi_{a_1, a_2, \dots, a_n}(R))$ .

### 15.1.5 I/O Cost for Scan Operators

As a simple application of the parameters that were introduced, we can represent the number of disk I/O's needed for each of the table-scan operators discussed so far. If relation  $R$  is clustered, then the number of disk I/O's for the table-scan operator is approximately  $B$ . Likewise, if  $R$  fits in main-memory, then we can implement sort-scan by reading  $R$  into memory and performing an in-memory sort, again requiring only  $B$  disk I/O's.

If  $R$  is clustered but requires a two-phase multiway merge sort, then, as discussed in Section 11.4.4, we require about  $3B$  disk I/O's, divided equally among the operations of reading  $R$  in sublists, writing out the sublists, and rereading the sublists. Remember that we do not charge for the final writing of the result. Neither do we charge memory space for accumulated output. Rather, we assume each output block is immediately consumed by some other operation; possibly it is simply written to disk.

However, if  $R$  is not clustered, then the number of required disk I/O's is generally much higher. If  $R$  is distributed among tuples of other relations, then a table-scan for  $R$  may require reading as many blocks as there are tuples of  $R$ ; that is, the I/O cost is  $T$ . Similarly, if we want to sort  $R$ , but  $R$  fits in memory, then  $T$  disk I/O's are what we need to get all of  $R$  into memory. Finally, if  $R$  is not clustered and requires a two-phase sort, then it takes  $T$  disk I/O's to read the subgroups initially. However, we may store and reread the sublists in clustered form, so these steps require only  $2B$  disk I/O's. The total cost for performing sort-scan on a large, unclustered relation is thus  $T + 2B$ .

Finally, let us consider the cost of an index-scan. Generally, an index on a relation  $R$  occupies many fewer than  $B(R)$  blocks. Therefore, a scan of the entire  $R$ , which takes at least  $B$  disk I/O's, will require significantly more I/O's than does examining the entire index. Thus, even though index-scan requires examining both the relation and its index,

- We continue to use  $B$  or  $T$  as an estimate of the cost of accessing a clustered or unclustered relation in its entirety, using an index.

### Why Iterators?

We shall see in Section 16.7 how iterators support efficient execution when they are composed within query plans. They contrast with a *materialization* strategy, where the result of each operator is produced in its entirety — and either stored on disk or allowed to take up space in main memory. When iterators are used, many operations are active at once. Tuples pass between operators as needed, thus reducing the need for storage. Of course, as we shall see, not all physical operators support the iteration approach, or “pipelining,” in a useful way. In some cases, almost all the work would need to be done by the `Open` function, which is tantamount to materialization.

However, if we only want part of  $R$ , we often are able to avoid looking at the entire index and the entire  $R$ . We shall defer analysis of these uses of indexes to Section 15.6.2.

#### 15.1.6 Iterators for Implementation of Physical Operators

Many physical operators can be implemented as an *iterator*, which is a group of three functions that allows a consumer of the result of the physical operator to get the result one tuple at a time. The three functions forming the iterator for an operation are:

1. `Open`. This function starts the process of getting tuples, but does not get a tuple. It initializes any data structures needed to perform the operation and calls `Open` for any arguments of the operation.
2. `GetNext`. This function returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained. In getting the next tuple of its result, it typically calls `GetNext` one or more times on its argument(s). If there are no more tuples to return, `GetNext` returns a special value `NotFound`, which we assume cannot be mistaken for a tuple.
3. `Close`. This function ends the iteration after all tuples, or all tuples that the consumer wanted, have been obtained. Typically, it calls `Close` on any arguments of the operator.

When describing iterators and their functions, we shall assume that there is a “class” for each type of iterator (i.e., for each type of physical operator implemented as an iterator), and the class supports `Open`, `GetNext`, and `Close` methods on instances of the class.

**Example 15.1:** Perhaps the simplest iterator is the one that implements the table-scan operator. The iterator is implemented by a class `TableScan`, and a table-scan operator in a query plan is an instance of this class parameterized by the relation  $R$  we wish to scan. Let us assume that  $R$  is a relation clustered in some list of blocks, which we can access in a convenient way; that is, the notion of “get the next block of  $R$ ” is implemented by the storage system and need not be described in detail. Further, we assume that within a block there is a directory of records (tuples) so that it is easy to get the next tuple of a block or tell that the last tuple has been reached.

```

Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
            RETURN NotFound;
        ELSE /* b is a new block */
            t := first tuple on block b;
    } /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

Close() {
}

```

Figure 15.3: Iterator functions for the table-scan operator over relation  $R$

Figure 15.3 sketches the three functions for this iterator. We imagine a block pointer  $b$  and a tuple pointer  $t$  that points to a tuple within block  $b$ . We assume that both pointers can point “beyond” the last block or last tuple of a block, respectively, and that it is possible to identify when these conditions occur. Notice that `Close` in this example does nothing. In practice, a `Close` function for an iterator might clean up the internal structure of the DBMS in various ways. It might inform the buffer manager that certain buffers are no longer needed, or inform the concurrency manager that the read of a relation has completed.  $\square$

**Example 15.2:** Now, let us consider an example where the iterator does most of the work in its `Open` function. The operator is sort-scan, where we read the

tuples of a relation  $R$  but return them in sorted order. Further, let us suppose that  $R$  is so large that we need to use a two-phase, multiway merge-sort, as in Section 11.4.4.

We cannot return even the first tuple until we have examined each tuple of  $R$ . Thus, `Open` must do at least the following:

1. Read all the tuples of  $R$  in main-memory-sized chunks, sort them, and store them on disk.
2. Initialize the data structure for the second (merge) phase, and load the first block of each sublist into the main-memory structure.

Then, `GetNext` can run a competition for the first remaining tuple at the heads of all the sublists. If the block from the winning sublist is exhausted, `GetNext` reloads its buffer.  $\square$

**Example 15.3:** Finally, let us consider a simple example of how iterators can be combined by calling other iterators. It is not a good example of how many iterators can be active simultaneously, but that will have to wait until we have considered algorithms for physical operators like selection and join, which exploit this capability of iterators better.

Our operation is the bag union  $R \cup S$ , in which we produce first all the tuples of  $R$  and then all the tuples of  $S$ , without regard for the existence of duplicates. Let  $\mathcal{R}$  and  $\mathcal{S}$  denote the iterators that produce relations  $R$  and  $S$ , and thus are the “children” of the union operator in a query plan for  $R \cup S$ . Iterators  $\mathcal{R}$  and  $\mathcal{S}$  could be table scans applied to stored relations  $R$  and  $S$ , or they could be iterators that call a network of other iterators to compute  $R$  and  $S$ . Regardless, all that is important is that we have available functions  $\mathcal{R}.\text{Open}$ ,  $\mathcal{R}.\text{GetNext}$ , and  $\mathcal{R}.\text{Close}$ , and analogous functions for iterator  $\mathcal{S}$ . The iterator functions for the union are sketched in Fig. 15.4. One subtle point is that the functions use a shared variable `CurRel` that is either  $\mathcal{R}$  or  $\mathcal{S}$ , depending on which relation is being read from currently.  $\square$

## 15.2 One-Pass Algorithms for Database Operations

We shall now begin our study of a very important topic in query optimization: how should we execute each of the individual steps — for example, a join or selection — of a logical query plan? The choice of an algorithm for each operator is an essential part of the process of transforming a logical query plan into a physical query plan. While many algorithms for operators have been proposed, they largely fall into three classes:

1. Sorting-based methods. These are covered primarily in Section 15.4.

```

Open() {
    R.Open();
    CurRel := R;
}

GetNext() {
    IF (CurRel = R) {
        t := R.GetNext();
        IF (t <> NotFound) /* R is not exhausted */
            RETURN t;
        ELSE /* R is exhausted */ {
            S.Open();
            CurRel := S;
        }
    }
    /* here, we must read from S */
    RETURN S.GetNext();
    /* notice that if S is exhausted, S.GetNext()
       will return NotFound, which is the correct
       action for our GetNext as well */
}

Close() {
    R.Close();
    S.Close();
}

```

Figure 15.4: Building a union iterator from iterators  $\mathcal{R}$  and  $\mathcal{S}$ 

2. Hash-based methods. These are mentioned in Section 15.5 and Section 15.9, among other places.
3. Index-based methods. These are emphasized in Section 15.6.

In addition, we can divide algorithms for operators into three “degrees” of difficulty and cost:

- a) Some methods involve reading the data only once from disk. These are the *one-pass* algorithms, and they are the topic of this section. Usually, they work only when at least one of the arguments of the operation fits in main memory, although there are exceptions, especially for selection and projection as discussed in Section 15.2.1.
- b) Some methods work for data that is too large to fit in available main memory but not for the largest imaginable data sets. An example of such

an algorithm is the two-phase, multiway merge sort of Section 11.4.4. These *two-pass* algorithms are characterized by reading data a first time from disk, processing it in some way, writing all, or almost all of it to disk, and then reading it a second time for further processing during the second pass. We meet these algorithms in Sections 15.4 and 15.5.

- c) Some methods work without a limit on the size of the data. These methods use three or more passes to do their jobs, and are natural, recursive generalizations of the two-pass algorithms; we shall study multipass methods in Section 15.8.

In this section, we shall concentrate on the one-pass methods. However, both in this section and subsequently, we shall classify operators into three broad groups:

1. *Tuple-at-a-time, unary operations.* These operations — selection and projection — do not require an entire relation, or even a large part of it, in memory at once. Thus, we can read a block at a time, use one main-memory buffer, and produce our output.
2. *Full-relation, unary operations.* These one-argument operations require seeing all or most of the tuples in memory at once, so one-pass algorithms are limited to relations that are approximately of size  $M$  (the number of main-memory buffers available) or less. The operations of this class that we consider here are  $\gamma$  (the grouping operator) and  $\delta$  (the duplicate-elimination operator).
3. *Full-relation, binary operations.* All other operations are in this class: set and bag versions of union, intersection, difference, joins, and products. Except for bag union, each of these operations requires at least one argument to be limited to size  $M$ , if we are to use a one-pass algorithm.

### 15.2.1 One-Pass Algorithms for Tuple-at-a-Time Operations

The tuple-at-a-time operations  $\sigma(R)$  and  $\pi(R)$  have obvious algorithms, regardless of whether the relation fits in main memory. We read the blocks of  $R$  one at a time into an input buffer, perform the operation on each tuple, and move the selected tuples or the projected tuples to the output buffer, as suggested by Fig. 15.5. Since the output buffer may be an input buffer of some other operator, or may be sending data to a user or application, we do not count the output buffer as needed space. Thus, we require only that  $M \geq 1$  for the input buffer, regardless of  $B$ .

The disk I/O requirement for this process depends only on how the argument relation  $R$  is provided. If  $R$  is initially on disk, then the cost is whatever it takes to perform a table-scan or index-scan of  $R$ . The cost was discussed in Section 15.1.5; typically it is  $B$  if  $R$  is clustered and  $T$  if it is not clustered.

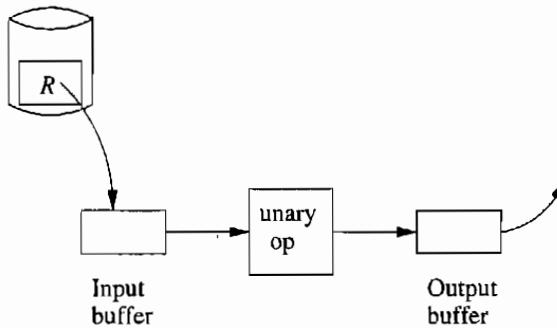


Figure 15.5: A selection or projection being performed on a relation  $R$

### Extra Buffers Can Speed Up Operations

Although tuple-at-a-time operations can get by with only one input buffer and one output buffer, as suggested by Fig. 15.5, we can often speed up processing if we allocate more input buffers. The idea appeared first in Section 11.5.1. If  $R$  is stored on consecutive blocks within cylinders, then we can read an entire cylinder into buffers, while paying for the seek time and rotational latency for only one block per cylinder. Similarly, if the output of the operation can be stored on full cylinders, we waste almost no time writing.

However, we should remind the reader again of the important exception when the operation being performed is a selection, and the condition compares a constant to an attribute that has an index. In that case, we can use the index to retrieve only a subset of the blocks holding  $R$ , thus improving performance, often markedly.

#### 15.2.2 One-Pass Algorithms for Unary, Full-Relation Operations

Now, let us consider the unary operations that apply to relations as a whole, rather than to one tuple at a time: duplicate elimination ( $\delta$ ) and grouping ( $\gamma$ ).

##### Duplicate Elimination

To eliminate duplicates, we can read each block of  $R$  one at a time, but for each tuple we need to make a decision as to whether:

1. It is the first time we have seen this tuple, in which case we copy it to the output, or

2. We have seen the tuple before, in which case we must not output this tuple.

To support this decision, we need to keep in memory one copy of every tuple we have seen, as suggested in Fig. 15.6. One memory buffer holds one block of  $R$ 's tuples, and the remaining  $M - 1$  buffers can be used to hold a single copy of every tuple seen so far.

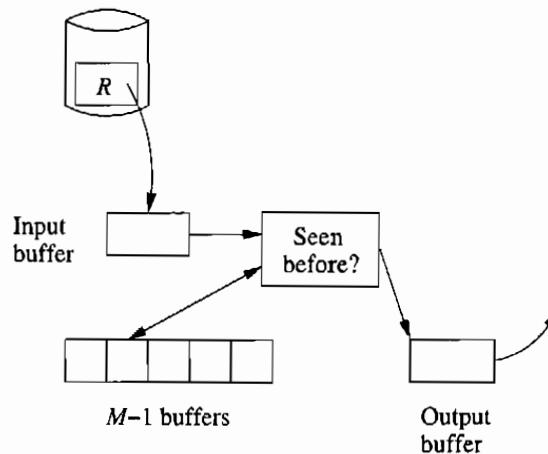


Figure 15.6: Managing memory for a one-pass duplicate-elimination

When storing the already-seen tuples, we must be careful about the main-memory data structure we use. Naively, we might just list the tuples we have seen. When a new tuple from  $R$  is considered, we compare it with all tuples seen so far, and if it is not equal to any of these tuples we both copy it to the output and add it to the in-memory list of tuples we have seen.

However, if there are  $n$  tuples in main memory, each new tuple takes processor time proportional to  $n$ , so the complete operation takes processor time proportional to  $n^2$ . Since  $n$  could be very large, this amount of time calls into serious question our assumption that only the disk I/O time is significant. Thus, we need a main-memory structure that allows each of the operations:

1. Add a new tuple, and
2. Tell whether a given tuple is already there

to be done in time that is close to a constant, independent of the number of tuples  $n$  that we currently have in memory. There are many such structures known. For example, we could use a hash table with a large number of buckets, or some form of balanced binary search tree.<sup>1</sup> Each of these structures has some

<sup>1</sup>See Aho, A. V., J. E. Hopcroft, and J. D. Ullman *Data Structures and Algorithms*, Addison-Wesley, 1984 for discussions of suitable main-memory structures. In particular, hashing takes on average  $O(n)$  time to process  $n$  items, and balanced trees take  $O(n \log n)$  time; either is sufficiently close to linear for our purposes.

space overhead in addition to the space needed to store the tuples; for instance, a main-memory hash table needs a bucket array and space for pointers to link the tuples in a bucket. However, the overhead tends to be small compared with the space needed to store the tuples. We shall thus make the simplifying assumption of no overhead space and concentrate on what is required to store the tuples in main memory.

On this assumption, we may store in the  $M - 1$  available buffers of main memory as many tuples as will fit in  $M - 1$  blocks of  $R$ . If we want one copy of each distinct tuple of  $R$  to fit in main memory, then  $B(\delta(R))$  must be no larger than  $M - 1$ . Since we expect  $M$  to be much larger than 1, a simpler approximation to this rule, and the one we shall generally use, is:

- $B(\delta(R)) \leq M$

Note that we cannot in general compute the size of  $\delta(R)$  without computing  $\delta(R)$  itself. Should we underestimate that size, so  $B(\delta(R))$  is actually larger than  $M$ , we shall pay a significant penalty due to thrashing, as the blocks holding the distinct tuples of  $R$  must be brought into and out of main memory frequently.

### Grouping

A grouping operation  $\gamma_L$  gives us zero or more grouping attributes and presumably one or more aggregated attributes. If we create in main memory one entry for each group — that is, for each value of the grouping attributes — then we can scan the tuples of  $R$ , one block at a time. The *entry* for a group consists of values for the grouping attributes and an accumulated value or values for each aggregation. The accumulated value is, except in one case, obvious:

- For a  $\text{MIN}(a)$  or  $\text{MAX}(a)$  aggregate, record the minimum or maximum value, respectively, of attribute  $a$  seen for any tuple in the group so far. Change this minimum or maximum, if appropriate, each time a tuple of the group is seen.
- For any  $\text{COUNT}$  aggregation, add one for each tuple of the group that is seen.
- For  $\text{SUM}(a)$ , add the value of attribute  $a$  to the accumulated sum for its group.
- $\text{AVG}(a)$  is the hard case. We must maintain two accumulations: the count of the number of tuples in the group and the sum of the  $a$ -values of these tuples. Each is computed as we would for a  $\text{COUNT}$  and  $\text{SUM}$  aggregation, respectively. After all tuples of  $R$  are seen, we take the quotient of the sum and count to obtain the average.

When all tuples of  $R$  have been read into the input buffer and contributed to the aggregation(s) for their group, we can produce the output by writing the tuple for each group. Note that until the last tuple is seen, we cannot begin to create output for a  $\gamma$  operation. Thus, this algorithm does not fit the iterator framework very well; the entire grouping has to be done by the `Open` function before the first tuple can be retrieved by `GetNext`.

In order that the in-memory processing of each tuple be efficient, we need to use a main-memory data structure that lets us find the entry for each group, given values for the grouping attributes. As discussed above for the  $\delta$  operation, common main-memory data structures such as hash tables or balanced trees will serve well. We should remember, however, that the search key for this structure is the grouping attributes only.

The number of disk I/O's needed for this one-pass algorithm is  $B$ , as must be the case for any one-pass algorithm for a unary operator. The number of required memory buffers  $M$  is not related to  $B$  in any simple way, although typically  $M$  will be less than  $B$ . The problem is that the entries for the groups could be longer or shorter than tuples of  $R$ , and the number of groups could be anything equal to or less than the number of tuples of  $R$ . However, in most cases, group entries will be no longer than  $R$ 's tuples, and there will be many fewer groups than tuples.

### 15.2.3 One-Pass Algorithms for Binary Operations

Let us now take up the binary operations: union, intersection, difference, product, and join. Since in some cases we must distinguish the set- and bag-versions of these operators, we shall subscript them with  $B$  or  $S$  for “bag” and “set,” respectively; e.g.,  $\cup_B$  for bag union or  $-_S$  for set difference. To simplify the discussion of joins, we shall consider only the natural join. An equijoin can be implemented the same way, after attributes are renamed appropriately, and theta-joins can be thought of as a product or equijoin followed by a selection for those conditions that cannot be expressed in an equijoin.

Bag union can be computed by a very simple one-pass algorithm. To compute  $R \cup_B S$ , we copy each tuple of  $R$  to the output and then copy every tuple of  $S$ , as we did in Example 15.3. The number of disk I/O's is  $B(R) + B(S)$ , as it must be for a one-pass algorithm on operands  $R$  and  $S$ , while  $M = 1$  suffices regardless of how large  $R$  and  $S$  are.

Other binary operations require reading the smaller of the operands  $R$  and  $S$  into main memory and building a suitable data structure so tuples can be both inserted quickly and found quickly, as discussed in Section 15.2.2. As before, a hash table or balanced tree suffices. The structure requires a small amount of space (in addition to the space for the tuples themselves), which we shall neglect. Thus, the approximate requirement for a binary operation on relations  $R$  and  $S$  to be performed in one pass is:

- $\min(B(R), B(S)) \leq M$

### Operations on Nonclustered Data

Remember that all our calculations regarding the number of disk I/O's required for an operation are predicated on the assumption that the operand relations are clustered. In the (typically rare) event that an operand  $R$  is not clustered, then it may take us  $T(R)$  disk I/O's, rather than  $B(R)$  disk I/O's to read all the tuples of  $R$ . Note, however, that any relation that is the result of an operator may always be assumed clustered, since we have no reason to store a temporary relation in a nonclustered fashion.

This rule assumes that one buffer will be used to read the blocks of the larger relation, while approximately  $M$  buffers are needed to house the entire smaller relation and its main-memory data structure.

We shall now give the details of the various operations. In each case, we assume  $R$  is the larger of the relations, and we house  $S$  in main memory.

#### Set Union

We read  $S$  into  $M - 1$  buffers of main memory and build a search structure where the search key is the entire tuple. All these tuples are also copied to the output. We then read each block of  $R$  into the  $M$ th buffer, one at a time. For each tuple  $t$  of  $R$ , we see if  $t$  is in  $S$ , and if not, we copy  $t$  to the output. If  $t$  is also in  $S$ , we skip  $t$ .

#### Set Intersection

Read  $S$  into  $M - 1$  buffers and build a search structure with full tuples as the search key. Read each block of  $R$ , and for each tuple  $t$  of  $R$ , see if  $t$  is also in  $S$ . If so, copy  $t$  to the output, and if not, ignore  $t$ .

#### Set Difference

Since difference is not commutative, we must distinguish between  $R -_S S$  and  $S -_R R$ , continuing to assume that  $R$  is the larger relation. In each case, read  $S$  into  $M - 1$  buffers and build a search structure with full tuples as the search key.

To compute  $R -_S S$ , we read each block of  $R$  and examine each tuple  $t$  on that block. If  $t$  is in  $S$ , then ignore  $t$ ; if it is not in  $S$  then copy  $t$  to the output.

To compute  $S -_R R$ , we again read the blocks of  $R$  and examine each tuple  $t$  in turn. If  $t$  is in  $S$ , then we delete  $t$  from the copy of  $S$  in main memory, while if  $t$  is not in  $S$  we do nothing. After considering each tuple of  $R$ , we copy to the output those tuples of  $S$  that remain.

### Bag Intersection

We read  $S$  into  $M - 1$  buffers, but we associate with each distinct tuple a *count*, which initially measures the number of times this tuple occurs in  $S$ . Multiple copies of a tuple  $t$  are not stored individually. Rather we store one copy of  $t$  and associate with it a count equal to the number of times  $t$  occurs.

This structure could take slightly more space than  $B(S)$  blocks if there were few duplicates, although frequently the result is that  $S$  is compacted. Thus, we shall continue to assume that  $B(S) \leq M$  is sufficient for a one-pass algorithm to work, although the condition is only an approximation.

Next, we read each block of  $R$ , and for each tuple  $t$  of  $R$  we see whether  $t$  occurs in  $S$ . If not we ignore  $t$ ; it cannot appear in the intersection. However, if  $t$  appears in  $S$ , and the count associated with  $t$  is still positive, then we output  $t$  and decrement the count by 1. If  $t$  appears in  $S$ , but its count has reached 0, then we do not output  $t$ ; we have already produced as many copies of  $t$  in the output as there were copies in  $S$ .

### Bag Difference

To compute  $S -_B R$ , we read the tuples of  $S$  into main memory, and count the number of occurrences of each distinct tuple, as we did for bag intersection. When we read  $R$ , for each tuple  $t$  we see whether  $t$  occurs in  $S$ , and if so, we decrement its associated count. At the end, we copy to the output each tuple in main memory whose count is positive, and the number of times we copy it equals that count.

To compute  $R -_B S$ , we also read the tuples of  $S$  into main memory and count the number of occurrences of distinct tuples. We may think of a tuple  $t$  with a count of  $c$  as  $c$  reasons not to copy  $t$  to the output as we read tuples of  $R$ . That is, when we read a tuple  $t$  of  $R$ , we see if  $t$  occurs in  $S$ . If not, then we copy  $t$  to the output. If  $t$  does occur in  $S$ , then we look at the current count  $c$  associated with  $t$ . If  $c = 0$ , then copy  $t$  to the output. If  $c > 0$ , do not copy  $t$  to the output, but decrement  $c$  by 1.

### Product

Read  $S$  into  $M - 1$  buffers of main memory; no special data structure is needed. Then read each block of  $R$ , and for each tuple  $t$  of  $R$  concatenate  $t$  with each tuple of  $S$  in main memory. Output each concatenated tuple as it is formed.

This algorithm may take a considerable amount of processor time per tuple of  $R$ , because each such tuple must be matched with  $M - 1$  blocks full of tuples. However, the output size is also large, and the time per output tuple is small.

### Natural Join

In this and other join algorithms, let us take the convention that  $R(X, Y)$  is being joined with  $S(Y, Z)$ , where  $Y$  represents all the attributes that  $R$  and  $S$

### What if $M$ is not Known?

While we present algorithms as if  $M$ , the number of available memory blocks, were fixed and known in advance, remember that the available  $M$  is often unknown, except within some obvious limits like the total memory of the machine. Thus, a query optimizer, when choosing between a one-pass and a two-pass algorithm, might estimate  $M$  and make the choice based on this estimate. If the optimizer is wrong, the penalty is either thrashing of buffers between disk and memory (if the guess of  $M$  was too high), or unnecessary passes if  $M$  was underestimated.

There are also some algorithms that degrade gracefully when there is less memory than expected. For example, we can behave like a one-pass algorithm, unless we run out of space, and then start behaving like a two-pass algorithm. Sections 15.5.6 and 15.7.3 discuss some of these approaches.

have in common,  $X$  is all attributes of  $R$  that are not in the schema of  $S$ , and  $Z$  is all attributes of  $S$  that are not in the schema of  $R$ . We continue to assume that  $S$  is the smaller relation. To compute the natural join, do the following:

1. Read all the tuples of  $S$  and form them into a main-memory search structure with the attributes of  $Y$  as the search key. As usual, a hash table or balanced tree are good examples of such structures. Use  $M - 1$  blocks of memory for this purpose.
2. Read each block of  $R$  into the one remaining main-memory buffer. For each tuple  $t$  of  $R$ , find the tuples of  $S$  that agree with  $t$  on all attributes of  $Y$ , using the search structure. For each matching tuple of  $S$ , form a tuple by joining it with  $t$ , and move the resulting tuple to the output.

Like all the one-pass, binary algorithms, this one takes  $B(R) + B(S)$  disk I/O's to read the operands. It works as long as  $B(S) \leq M - 1$ , or approximately,  $B(S) \leq M$ . Also as for the other algorithms we have studied, the space required by the main-memory search structure is not counted but may lead to a small, additional memory requirement.

We shall not discuss joins other than the natural join. Remember that an equijoin is executed in essentially the same way as a natural join, but we must account for the fact that “equal” attributes from the two relations may have different names. A theta-join that is not an equijoin can be replaced by an equijoin or product followed by a selection.

### 15.2.4 Exercises for Section 15.2

**Exercise 15.2.1:** For each of the operations below, write an iterator that uses the algorithm described in this section.

- \* a) Projection.
- \* b) Distinct ( $\delta$ ).
- c) Grouping ( $\gamma_L$ ).
- \* d) Set union.
- e) Set intersection.
- f) Set difference.
- g) Bag intersection.
- h) Bag difference.
- i) Product.
- j) Natural join.

**Exercise 15.2.2:** For each of the operators in Exercise 15.2.1, tell whether the operator is *blocking*, by which we mean that the first output cannot be produced until all the input has been read. Put another way, a blocking operator is one whose only possible iterators have all the important work done by `Open`.

**Exercise 15.2.3:** Figure 15.9 summarizes the memory and disk-I/O requirements of the algorithms of this section and the next. However, it assumes all arguments are clustered. How would the entries change if one or both arguments were not clustered?

! **Exercise 15.2.4:** Give one-pass algorithms for each of the following join-like operators:

- \* a)  $R \bowtie S$ , assuming  $R$  fits in memory (see Exercise 5.2.10 for a definition of the semijoin).
- \* b)  $R \bowtie S$ , assuming  $S$  fits in memory.
- c)  $R \overline{\bowtie} S$ , assuming  $R$  fits in memory (see Exercise 5.2.11 for a definition of the antisemijoin).
- d)  $R \overline{\bowtie} S$ , assuming  $S$  fits in memory.
- \* e)  $R \circledast_L S$ , assuming  $R$  fits in memory (see Section 5.4.7 for definitions involving outerjoins).
- f)  $R \bowtie_L S$ , assuming  $S$  fits in memory.

- g)  $R \bowtie_R S$ , assuming  $R$  fits in memory.
- h)  $R \bowtie_R S$ , assuming  $S$  fits in memory.
- i)  $R \bowtie S$ , assuming  $R$  fits in memory.

## 15.3 Nested-Loop Joins

Before proceeding to the more complex algorithms in the next sections, we shall turn our attention to a family of algorithms for the join operator called “nested-loop” joins. These algorithms are, in a sense, “one-and-a-half” passes, since in each variation one of the two arguments has its tuples read only once, while the other argument will be read repeatedly. Nested-loop joins can be used for relations of any size; it is not necessary that one relation fit in main memory.

### 15.3.1 Tuple-Based Nested-Loop Join

The simplest variation of nested-loop join has loops that range over individual tuples of the relations involved. In this algorithm, which we call *tuple-based nested-loop join*, we compute the join  $R(X, Y) \bowtie S(Y, Z)$  as follows:

```

FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t;
    
```

If we are careless about how we buffer the blocks of relations  $R$  and  $S$ , then this algorithm could require as many as  $T(R)T(S)$  disk I/O’s. However, there are many situations where this algorithm can be modified to have much lower cost. One case is when we can use an index on the join attribute or attributes of  $R$  to find the tuples of  $R$  that match a given tuple of  $S$ , without having to read the entire relation  $R$ . We discuss index-based joins in Section 15.6.3. A second improvement looks much more carefully at the way tuples of  $R$  and  $S$  are divided among blocks, and uses as much of the memory as it can to reduce the number of disk I/O’s as we go through the inner loop. We shall consider this block-based version of nested-loop join in Section 15.3.3.

### 15.3.2 An Iterator for Tuple-Based Nested-Loop Join

One advantage of a nested-loop join is that it fits well into an iterator framework, and thus, as we shall see in Section 16.7.3, allows us to avoid storing intermediate relations on disk in some situations. The iterator for  $R \bowtie S$  is easy to build from the iterators for  $R$  and  $S$ , which support functions  $R.\text{Open}()$ , and so on, as in Section 15.1.6. The code for the three iterator functions for nested-loop join is in Fig. 15.7. It makes the assumption that neither relation  $R$  nor  $S$  is empty.

```

Open() {
    R.Open();
    S.Open();
    s := S.GetNext();
}

GetNext() {
    REPEAT {
        r := R.GetNext();
        IF (r = NotFound) { /* R is exhausted for
                           the current s */
            R.Close();
            s := S.GetNext();
            IF (s = NotFound) RETURN NotFound;
            /* both R and S are exhausted */
            R.Open();
            r := R.GetNext();
        }
    }
    UNTIL(r and s join);
    RETURN the join of r and s;
}

Close() {
    R.Close();
    S.Close();
}

```

Figure 15.7: Iterator functions for tuple-based nested-loop join of  $R$  and  $S$

### 15.3.3 A Block-Based Nested-Loop Join Algorithm

We can improve on the tuple-based nested-loop join of Section 15.3.1 if we compute  $R \bowtie S$  by:

1. Organizing access to both argument relations by blocks, and
2. Using as much main memory as we can to store tuples belonging to the relation  $S$ , the relation of the outer loop.

Point (1) makes sure that when we run through the tuples of  $R$  in the inner loop, we use as few disk I/O's as possible to read  $R$ . Point (2) enables us to join each tuple of  $R$  that we read with not just one tuple of  $S$ , but with as many tuples of  $S$  as will fit in memory.

As in Section 15.2.3, let us assume  $B(S) \leq B(R)$ , but now let us also assume that  $B(S) > M$ ; i.e., neither relation fits entirely in main memory. We repeatedly read  $M-1$  blocks of  $S$  into main-memory buffers. A search structure, with search key equal to the common attributes of  $R$  and  $S$ , is created for the tuples of  $S$  that are in main memory. Then we go through all the blocks of  $R$ , reading each one in turn into the last block of memory. Once there, we compare all the tuples of  $R$ 's block with all the tuples in all the blocks of  $S$  that are currently in main memory. For those that join, we output the joined tuple. The nested-loop structure of this algorithm can be seen when we describe the algorithm more formally, in Fig. 15.8.

```

FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;

```

Figure 15.8: The nested-loop join algorithm

The program of Fig. 15.8 appears to have three nested loops. However, there really are only two loops if we look at the code at the right level of abstraction. The first, or outer loop, runs through the tuples of  $S$ . The other two loops run through the tuples of  $R$ . However, we expressed the process as two loops to emphasize that the order in which we visit the tuples of  $R$  is not arbitrary. Rather, we need to look at these tuples a block at a time (the role of the second loop), and within one block, we look at all the tuples of that block before moving on to the next block (the role of the third loop).

**Example 15.4:** Let  $B(R) = 1000$ ,  $B(S) = 500$ , and  $M = 101$ . We shall use 100 blocks of memory to buffer  $S$  in 100-block chunks, so the outer loop of Fig. 15.8 iterates five times. At each iteration, we do 100 disk I/O's to read the chunk of  $S$ , and we must read  $R$  entirely in the second loop, using 1000 disk I/O's. Thus, the total number of disk I/O's is 5500.

Notice that if we reversed the roles of  $R$  and  $S$ , the algorithm would use slightly more disk I/O's. We would iterate 10 times through the outer loop and do 600 disk I/O's at each iteration, for a total of 6000. In general, there is a slight advantage to using the smaller relation in the outer loop.  $\square$

The algorithm of Fig. 15.8 is sometimes called “nested-block join.” We shall continue to call it simply *nested-loop join*, since it is the variant of the nested-loop idea most commonly implemented in practice. If necessary to distinguish it from the tuple-based nested-loop join of Section 15.3.1, we can call Fig. 15.8 “block-based nested-loop join.”

#### 15.3.4 Analysis of Nested-Loop Join

The analysis of Example 15.4 can be repeated for any  $B(R)$ ,  $B(S)$ , and  $M$ . Assuming  $S$  is the smaller relation, the number of chunks, or iterations of the outer loop is  $B(S)/(M - 1)$ . At each iteration, we read  $M - 1$  blocks of  $S$  and  $B(R)$  blocks of  $R$ . The number of disk I/O’s is thus

$$\frac{B(S)}{M - 1} (M - 1 + B(R))$$

or

$$B(S) + \frac{B(S)B(R)}{M - 1}$$

Assuming all of  $M$ ,  $B(S)$ , and  $B(R)$  are large, but  $M$  is the smallest of these, an approximation to the above formula is  $B(S)B(R)/M$ . That is, the cost is proportional to the product of the sizes of the two relations, divided by the amount of available main memory. We can do much better than a nested-loop join when both relations are large. But for reasonably small examples such as Example 15.4, the cost of the nested-loop join is not much greater than the cost of a one-pass join, which is 1500 disk I/O’s for this example. In fact, if  $B(S) \leq M - 1$ , the nested-loop join becomes identical to the one-pass join algorithm of Section 15.2.3.

Although nested-loop join is generally not the most efficient join algorithm possible, we should note that in some early relational DBMS’s, it was the only method available. Even today, it is needed as a subroutine in more efficient join algorithms in certain situations, such as when large numbers of tuples from each relation share a common value for the join attribute(s). For an example where nested-loop join is essential, see Section 15.4.5.

#### 15.3.5 Summary of Algorithms so Far

The main-memory and disk I/O requirements for the algorithms we have discussed in Sections 15.2 and 15.3 are shown in Fig. 15.9. The memory requirements for  $\gamma$  and  $\delta$  are actually more complex than shown, and  $M = B$  is only a loose approximation. For  $\gamma$ ,  $M$  grows with the number of groups, and for  $\delta$ ,  $M$  grows with the number of distinct tuples.

#### 15.3.6 Exercises for Section 15.3

**Exercise 15.3.1:** Give the three iterator functions for the block-based version of nested-loop join.

Operators	Approximate $M$ required	Disk I/O	Section
$\sigma, \pi$	1	$B$	15.2.1
$\gamma, \delta$	$B$	$B$	15.2.2
$\cup, \cap, -, \times, \bowtie$	$\min(B(R), B(S))$	$B(R) + B(S)$	15.2.3
$\bowtie$	any $M \geq 2$	$B(R)B(S)/M$	15.3.3

Figure 15.9: Main memory and disk I/O requirements for one-pass and nested-loop algorithms

\* **Exercise 15.3.2:** Suppose  $B(R) = B(S) = 10,000$ , and  $M = 1000$ . Calculate the disk I/O cost of a nested-loop join.

**Exercise 15.3.3:** For the relations of Exercise 15.3.2, what value of  $M$  would we need to compute  $R \bowtie S$  using the nested-loop algorithm with no more than a) 100,000 ! b) 25,000 ! c) 15,000 disk I/O's?

! **Exercise 15.3.4:** If  $R$  and  $S$  are both unclustered, it seems that nested-loop join would require about  $T(R)T(S)/M$  disk I/O's.

- a) How can you do significantly better than this cost?
- b) If only one of  $R$  and  $S$  is unclustered, how would you perform a nested-loop join? Consider both the cases that the larger is unclustered and that the smaller is unclustered.

! **Exercise 15.3.5:** The iterator of Fig. 15.7 will not work properly if either  $R$  or  $S$  is empty. Rewrite the functions so they will work, even if one or both relations are empty.

## 15.4 Two-Pass Algorithms Based on Sorting

We shall now begin the study of multipass algorithms for performing relational-algebra operations on relations that are larger than what the one-pass algorithms of Section 15.2 can handle. We concentrate on *two-pass algorithms*, where data from the operand relations is read into main memory, processed in some way, written out to disk again, and then reread from disk to complete the operation. We can naturally extend this idea to any number of passes, where the data is read many times into main memory. However, we concentrate on two-pass algorithms because:

- a) Two passes are usually enough, even for very large relations,
- b) Generalizing to more than two passes is not hard; we discuss these extensions in Section 15.8.

In this section, we consider sorting as a tool for implementing relational operations. The basic idea is as follows. If we have a large relation  $R$ , where  $B(R)$  is larger than  $M$ , the number of memory buffers we have available, then we can repeatedly:

1. Read  $M$  blocks of  $R$  into main memory.
2. Sort these  $M$  blocks in main memory, using an efficient, main-memory sorting algorithm. Such an algorithm will take an amount of processor time that is just slightly more than linear in the number of tuples in main memory, so we expect that the time to sort will not exceed the disk I/O time for step (1).
3. Write the sorted list into  $M$  blocks of disk. We shall refer to the contents of these blocks as one of the *sorted sublists* of  $R$ .

All the algorithms we shall discuss then use a second pass to “merge” the sorted sublists in some way to execute the desired operator.

#### 15.4.1 Duplicate Elimination Using Sorting

To perform the  $\delta(R)$  operation in two passes, we sort the tuples of  $R$  in sublists as described above. We then use the available main memory to hold one block from each sorted sublist, as we did for the multiway merge sort of Section 11.4.4. However, instead of sorting the tuples from these sublists, we repeatedly copy one to the output and ignore all tuples identical to it. The process is suggested by Fig. 15.10.

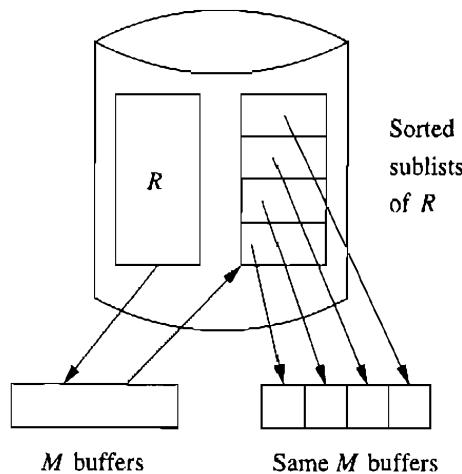


Figure 15.10: A two-pass algorithm for eliminating duplicates

More precisely, we look at the first unconsidered tuple from each block, and we find among them the first in sorted order, say  $t$ . We make one copy of  $t$  in

the output, and we remove from the fronts of the various input blocks all copies of  $t$ . If a block is exhausted, we bring into its buffer the next block from the same sublist, and if there are  $t$ 's on that block we remove them as well.

**Example 15.5:** Suppose for simplicity that tuples are integers, and only two tuples fit on a block. Also,  $M = 3$ ; i.e., there are three blocks in main memory. The relation  $R$  consists of 17 tuples:

$$2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3$$

We read the first six tuples into the three blocks of main memory, sort them, and write them out as the sublist  $R_1$ . Similarly, tuples seven through twelve are then read in, sorted and written as the sublist  $R_2$ . The last five tuples are likewise sorted and become the sublist  $R_3$ .

To start the second pass, we can bring into main memory the first block (two tuples) from each of the three sublists. The situation is now:

Sublist	In memory	Waiting on disk
$R_1$ :	1 2	2 2, 2 5
$R_2$ :	2 3	4 4, 4 5
$R_3$ :	1 1	2 3, 5

Looking at the first tuples of the three blocks in main memory, we find that 1 is the first tuple in sorted order. We therefore make one copy of 1 on the output, and we remove all 1's from the blocks in memory. When we do so, the block from  $R_3$  is exhausted, so we bring in the next block, with tuples 2 and 3, from that sublist. Had there been more 1's on this block, we would eliminate them. The situation is now:

Sublist	In memory	Waiting on disk
$R_1$ :	2	2 2, 2 5
$R_2$ :	2 3	4 4, 4 5
$R_3$ :	2 3	5

Now, 2 is the least tuple at the fronts of the lists, and in fact it happens to appear on each list. We write one copy of 2 to the output and eliminate 2's from the in-memory blocks. The block from  $R_1$  is exhausted and the next block from that sublist is brought to memory. That block has 2's, which are eliminated, again exhausting the block from  $R_1$ . The third block from that sublist is brought to memory, and its 2 is eliminated. The present situation is:

Sublist	In memory	Waiting on disk
$R_1$ :	5	
$R_2$ :	3	4 4, 4 5
$R_3$ :	3	5

Now, 3 is selected as the least tuple, one copy of 3 is written to the output, and the blocks from  $R_2$  and  $R_3$  are exhausted and replaced from disk, leaving:

Sublist	In memory	Waiting on disk
$R_1:$	5	
$R_2:$	4 4	4 5
$R_3:$	5	

To complete the example, 4 is next selected, consuming most of list  $R_2$ . At the final step, each list happens to consist of a single 5, which is output once and eliminated from the input buffers.  $\square$

The number of disk I/O's performed by this algorithm, as always ignoring the handling of the output, is:

1.  $B(R)$  to read each block of  $R$  when creating the sorted sublists.
2.  $B(R)$  to write each of the sorted sublists to disk.
3.  $B(R)$  to read each block from the sublists at the appropriate time.

Thus, the total cost of this algorithm is  $3B(R)$ , compared with  $B(R)$  for the single-pass algorithm of Section 15.2.2.

On the other hand, we can handle much larger files using the two-pass algorithm than with the one-pass algorithm. Assuming  $M$  blocks of memory are available, we create sorted sublists of  $M$  blocks each. For the second pass, we need one block from each sublist in main memory, so there can be no more than  $M$  sublists, each  $M$  blocks long. Thus,  $B \leq M^2$  is required for the two-pass algorithm to be feasible, compared with  $B \leq M$  for the one-pass algorithm. Put another way, to compute  $\delta(R)$  with the two-pass algorithm requires only  $\sqrt{B(R)}$  blocks of main memory, rather than  $B(R)$  blocks.

### 15.4.2 Grouping and Aggregation Using Sorting

The two-pass algorithm for  $\gamma_L(R)$  is quite similar to the algorithm of Section 15.4.1 for  $\delta(R)$ . We summarize it as follows:

1. Read the tuples of  $R$  into memory,  $M$  blocks at a time. Sort each  $M$  blocks, using the grouping attributes of  $L$  as the sort key. Write each sorted sublist to disk.
2. Use one main-memory buffer for each sublist, and initially load the first block of each sublist into its buffer.
3. Repeatedly find the least value of the sort key (grouping attributes) present among the first available tuples in the buffers. This value,  $v$ , becomes the next group, for which we:
  - (a) Prepare to compute all the aggregates on list  $L$  for this group. As in Section 15.2.2, use a count and sum in place of an average.

- (b) Examine each of the tuples with sort key  $v$ , and accumulate the needed aggregates.
- (c) If a buffer becomes empty, replace it with the next block from the same sublist.

When there are no more tuples with sort key  $v$  available, output a tuple consisting of the grouping attributes of  $L$  and the associated values of the aggregations we have computed for the group.

As for the  $\delta$  algorithm, this two-pass algorithm for  $\gamma$  takes  $3B(R)$  disk I/O's, and will work as long as  $B(R) \leq M^2$ .

#### 15.4.3 A Sort-Based Union Algorithm

When bag-union is wanted, the one-pass algorithm of Section 15.2.3, where we simply copy both relations, works regardless of the size of the arguments, so there is no need to consider a two-pass algorithm for  $\cup_B$ . However, the one-pass algorithm for  $\cup_S$  only works when at least one relation is smaller than the available main memory, so we should consider a two-pass algorithm for set union. The methodology we present works for the set and bag versions of intersection and difference as well, as we shall see in Section 15.4.4. To compute  $R \cup_S S$ , we do the following:

1. Repeatedly bring  $M$  blocks of  $R$  into main memory, sort their tuples, and write the resulting sorted sublist back to disk.
2. Do the same for  $S$ , to create sorted sublists for relation  $S$ .
3. Use one main-memory buffer for each sublist of  $R$  and  $S$ . Initialize each with the first block from the corresponding sublist.
4. Repeatedly find the first remaining tuple  $t$  among all the buffers. Copy  $t$  to the output, and remove from the buffers all copies of  $t$  (if  $R$  and  $S$  are sets there should be at most two copies). If a buffer becomes empty, reload it with the next block from its sublist.

We observe that each tuple of  $R$  and  $S$  is read twice into main memory, once when the sublists are being created, and the second time as part of one of the sublists. The tuple is also written to disk once, as part of a newly formed sublist. Thus, the cost in disk I/O's is  $3(B(R) + B(S))$ .

The algorithm works as long as the total number of sublists among the two relations does not exceed  $M$ , because we need one buffer for each sublist. Since each sublist is  $M$  blocks long, that says the sizes of the two relations must not exceed  $M^2$ ; that is,  $B(R) + B(S) \leq M^2$ .

### 15.4.4 Sort-Based Intersection and Difference

Whether the set version or the bag version is wanted, the algorithms are essentially the same as that of Section 15.4.3, except that the way we handle the copies of a tuple  $t$  at the fronts of the sorted sublists differs. In general we create the sorted sublists of  $M$  blocks each for both argument relations  $R$  and  $S$ . We use one main-memory buffer for each sublist, initially loaded with the first block of the sublist.

We then repeatedly consider the least tuple  $t$  among the remaining tuples in all the buffers. We count the number of tuples of  $R$  that are identical to  $t$  and we also count the number of tuples of  $S$  that are identical to  $t$ . Doing so requires that we reload buffers from any sublists whose currently buffered block is exhausted. The following indicates how we determine whether  $t$  is output, and if so, how many times:

- For set intersection, output  $t$  if it appears in both  $R$  and  $S$ .
- For bag intersection, output  $t$  the minimum of the number of times it appears in  $R$  and in  $S$ . Note that  $t$  is not output if either of these counts is 0; that is, if  $t$  is missing from one or both of the relations.
- For set difference,  $R -_S S$ , output  $t$  if and only if it appears in  $R$  but not in  $S$ .
- For bag difference,  $R -_B S$ , output  $t$  the number of times it appears in  $R$  minus the number of times it appears in  $S$ . Of course, if  $t$  appears in  $S$  at least as many times as it appears in  $R$ , then do not output  $t$  at all.

**Example 15.6:** Let us make the same assumptions as in Example 15.5:  $M = 3$ , tuples are integers, and two tuples fit in a block. The data will be almost the same as in that example as well. However, here we need two arguments, so we shall assume that  $R$  has 12 tuples and  $S$  has 5 tuples. Since main memory can fit six tuples, in the first pass we get two sublists from  $R$ , which we shall call  $R_1$  and  $R_2$ , and only one sorted sublist from  $S$ , which we refer to as  $S_1$ .<sup>2</sup> After creating the sorted sublists (from unsorted relations similar to the data from Example 15.5), the situation is:

Sublist	In memory	Waiting on disk
$R_1$ :	1 2	2 2, 2 5
$R_2$ :	2 3	4 4, 4 5
$S_1$ :	1 1	2 3, 5

Suppose we want to take the bag difference  $R -_B S$ . We find that the least tuple among the main-memory buffers is 1, so we count the number of 1's among the sublists of  $R$  and among the sublists of  $S$ . We find that 1 appears once in  $R$

---

<sup>2</sup>Since  $S$  fits in main memory, we could actually use the one-pass algorithms of Section 15.2.3, but we shall use the two-pass approach for illustration.

and twice in  $S$ . Since 1 does not appear more times in  $R$  than in  $S$ , we do not output any copies of tuple 1. Since the first block of  $S_1$  was exhausted counting 1's, we loaded the next block of  $S_1$ , leaving the following situation:

Sublist	In memory	Waiting on disk
$R_1$ :	2	2 2, 2 5
$R_2$ :	2 3	4 4, 4 5
$S_1$ :	2 3	5

We now find that 2 is the least remaining tuple, so we count the number of its occurrences in  $R$ , which is five occurrences, and we count the number of its occurrences in  $S$ , which is one. We thus output tuple 2 four times. As we perform the counts, we must reload the buffer for  $R_1$  twice, which leaves:

Sublist	In memory	Waiting on disk
$R_1$ :	5	
$R_2$ :	3	4 4, 4 5
$S_1$ :	3	5

Next, we consider tuple 3, and find it appears once in  $R$  and once in  $S$ . We therefore do not output 3 and remove its copies from the buffers, leaving:

Sublist	In memory	Waiting on disk
$R_1$ :	5	
$R_2$ :	4 4	4 5
$S_1$ :	5	

Tuple 4 occurs three times in  $R$  and not at all in  $S$ , so we output three copies of 4. Last, 5 appears twice in  $R$  and once in  $S$ , so we output 5 once. The complete output is 2, 2, 2, 2, 4, 4, 4, 5.  $\square$

The analysis of this family of algorithms is the same as for the set-union algorithm described in Section 15.4.3:

- $3(B(R) + B(S))$  disk I/O's.
- Approximately  $B(R) + B(S) \leq M^2$  for the algorithm to work.

#### 15.4.5 A Simple Sort-Based Join Algorithm

There are several ways that sorting can be used to join large relations. Before examining the join algorithms, let us observe one problem that can occur when we compute a join but was not an issue for the binary operations considered so far. When taking a join, the number of tuples from the two relations that share a common value of the join attribute(s), and therefore need to be in main memory simultaneously, can exceed what fits in memory. The extreme example is when there is only one value of the join attribute(s), and every tuple of one

relation joins with every tuple of the other relation. In this situation, there is really no choice but to take a nested-loop join of the two sets of tuples with a common value in the join-attribute(s).

To avoid facing this situation, we can try to reduce main-memory use for other aspects of the algorithm, and thus make available a large number of buffers to hold the tuples with a given join-attribute value. In this section we shall discuss the algorithm that makes the greatest possible number of buffers available for joining tuples with a common value. In Section 15.4.7 we consider another sort-based algorithm that uses fewer disk I/O's, but can present problems when there are large numbers of tuples with a common join-attribute value.

Given relations  $R(X, Y)$  and  $S(Y, Z)$  to join, and given  $M$  blocks of main memory for buffers, we do the following:

1. Sort  $R$ , using a two-phase, multiway merge sort, with  $Y$  as the sort key.
2. Sort  $S$  similarly.
3. Merge the sorted  $R$  and  $S$ . We generally use only two buffers, one for the current block of  $R$  and the other for the current block of  $S$ . The following steps are done repeatedly:
  - (a) Find the least value  $y$  of the join attributes  $Y$  that is currently at the front of the blocks for  $R$  and  $S$ .
  - (b) If  $y$  does not appear at the front of the other relation, then remove the tuple(s) with sort key  $y$ .
  - (c) Otherwise, identify all the tuples from both relations having sort key  $y$ . If necessary, read blocks from the sorted  $R$  and/or  $S$ , until we are sure there are no more  $y$ 's in either relation. As many as  $M$  buffers are available for this purpose.
  - (d) Output all the tuples that can be formed by joining tuples from  $R$  and  $S$  with a common  $Y$ -value  $y$ .
  - (e) If either relation has no more unconsidered tuples in main memory, reload the buffer for that relation.

**Example 15.7:** Let us consider the relations  $R$  and  $S$  from Example 15.4. Recall these relations occupy 1000 and 500 blocks, respectively, and there are  $M = 101$  main-memory buffers. When we use two-phase, multiway merge sort on a relation, we do four disk I/O's per block, two in each of the two phases. Thus, we use  $4(B(R) + B(S))$  disk I/O's to sort  $R$  and  $S$ , or 6000 disk I/O's.

When we merge the sorted  $R$  and  $S$  to find the joined tuples, we read each block of  $R$  and  $S$  a fifth time, using another 1500 disk I/O's. In this merge we generally need only two of the 101 blocks of memory. However, if necessary, we could use all 101 blocks to hold the tuples of  $R$  and  $S$  that share a common  $Y$ -value  $y$ . Thus, it is sufficient that for no  $y$  do the tuples of  $R$  and  $S$  that have  $Y$ -value  $y$  together occupy more than 101 blocks.

Notice that the total number of disk I/O's performed by this algorithm is 7500, compared with 5500 for nested-loop join in Example 15.4. However, nested-loop join is inherently a quadratic algorithm, taking time proportional to  $B(R)B(S)$ , while sort-join has linear I/O cost, taking time proportional to  $B(R) + B(S)$ . It is only the constant factors and the small size of the example (each relation is only 5 or 10 times larger than a relation that fits entirely in the allotted buffers) that make nested-loop join preferable. Moreover, we shall see in Section 15.4.7 that it is usually possible to perform a sort-join in  $3(B(R) + B(S))$  disk I/O's, which would be 4500 in this example and which is below the cost of nested-loop join.  $\square$

If there is a  $Y$ -value  $y$  for which the number of tuples with this  $Y$ -value does not fit in  $M$  buffers, then we need to modify the above algorithm.

1. If the tuples from one of the relations, say  $R$ , that have  $Y$ -value  $y$  fit in  $M - 1$  buffers, then load these blocks of  $R$  into buffers, and read the blocks of  $S$  that hold tuples with  $y$ , one at a time, into the remaining buffer. In effect, we do the one-pass join of Section 15.2.3 on only the tuples with  $Y$ -value  $y$ .
2. If neither relation has sufficiently few tuples with  $Y$ -value  $y$  that they all fit in  $M - 1$  buffers, then use the  $M$  buffers to perform a nested-loop join on the tuples with  $Y$ -value  $y$  from both relations.

Note that in either case, it may be necessary to read blocks from one relation and then ignore them, having to read them later. For example, in case (1), we might first read the blocks of  $S$  that have tuples with  $Y$ -value  $y$  and find that there are too many to fit in  $M - 1$  buffers. However, if we then read the tuples of  $R$  with that  $Y$ -value we find that they do fit in  $M - 1$  buffers.

#### 15.4.6 Analysis of Simple Sort-Join

As we noted in Example 15.7, our algorithm performs five disk I/O's for every block of the argument relation. The exception would be if there were so many tuples with a common  $Y$ -value that we needed to do one of the specialized joins on these tuples. In that case, the number of extra disk I/O's depends on whether one or both relations have so many tuples with a common  $Y$ -value that they require more than  $M - 1$  buffers by themselves. We shall not go into all the cases here; the exercises contain some examples to work out.

We also need to consider how big  $M$  needs to be in order for the simple sort-join to work. The primary constraint is that we need to be able to perform the two-phase, multiway merge sorts on  $R$  and  $S$ . As we observed in Section 11.4.4, we need  $B(R) \leq M^2$  and  $B(S) \leq M^2$  to perform these sorts. Once done, we shall not run out of buffers, although as discussed before, we may have to deviate from the simple merge if the tuples with a common  $Y$ -value cannot fit in  $M$  buffers. In summary, assuming no such deviations are necessary:

- The simple sort-join uses  $5(B(R) + B(S))$  disk I/O's.
- It requires  $B(R) \leq M^2$  and  $B(S) \leq M^2$  to work.

### 15.4.7 A More Efficient Sort-Based Join

If we do not have to worry about very large numbers of tuples with a common value for the join attribute(s), then we can save two disk I/O's per block by combining the second phase of the sorts with the join itself. We call this algorithm *sort-join*; other names by which it is known include “merge-join” and “sort-merge-join.” To compute  $R(X, Y) \bowtie S(Y, Z)$  using  $M$  main-memory buffers:

1. Create sorted sublists of size  $M$ , using  $Y$  as the sort key, for both  $R$  and  $S$ .
2. Bring the first block of each sublist into a buffer; we assume there are no more than  $M$  sublists in all.
3. Repeatedly find the least  $Y$ -value  $y$  among the first available tuples of all the sublists. Identify all the tuples of both relations that have  $Y$ -value  $y$ , perhaps using some of the  $M$  available buffers to hold them, if there are fewer than  $M$  sublists. Output the join of all tuples from  $R$  with all tuples from  $S$  that share this common  $Y$ -value. If the buffer for one of the sublists is exhausted, then replenish it from disk.

**Example 15.8:** Let us again consider the problem of Example 15.4: joining relations  $R$  and  $S$  of sizes 1000 and 500 blocks, respectively, using 101 buffers. We divide  $R$  into 10 sublists and  $S$  into 5 sublists, each of length 100, and sort them.<sup>3</sup> We then use 15 buffers to hold the current blocks of each of the sublists. If we face a situation in which many tuples have a fixed  $Y$ -value, we can use the remaining 86 buffers to store these tuples, but if there are more tuples than that we must use a special algorithm such as was discussed at the end of Section 15.4.5.

Assuming that we do not need to modify the algorithm for large groups of tuples with the same  $Y$ -value, then we perform three disk I/O's per block of data. Two of those are to create the sorted sublists. Then, every block of every sorted sublist is read into main memory one more time in the multiway merging process. Thus, the total number of disk I/O's is 4500.  $\square$

This sort-join algorithm is more efficient than the algorithm of Section 15.4.5 when it can be used. As we observed in Example 15.8, the number of disk I/O's is  $3(B(R) + B(S))$ . We can perform the algorithm on data that is almost as large as that of the previous algorithm. The sizes of the sorted sublists are

---

<sup>3</sup>Technically, we could have arranged for the sublists to have length 101 blocks each, with the last sublist of  $R$  having 91 blocks and the last sublist of  $S$  having 96 blocks, but the costs would turn out exactly the same.

$M$  blocks, and there can be at most  $M$  of them among the two lists. Thus,  $B(R) + B(S) \leq M^2$  is sufficient.

We might wonder whether we can avoid the trouble that arises when there are many tuples with a common  $Y$ -value. Some important considerations are:

1. Sometimes we can be sure the problem will not arise. For example, if  $Y$  is a key for  $R$ , then a given  $Y$ -value  $y$  can appear only once among all the blocks of the sublists for  $R$ . When it is  $y$ 's turn, we can leave the tuple from  $R$  in place and join it with all the tuples of  $S$  that match. If blocks of  $S$ 's sublists are exhausted during this process, they can have their buffers reloaded with the next block, and there is never any need for additional space, no matter how many tuples of  $S$  have  $Y$ -value  $y$ . Of course, if  $Y$  is a key for  $S$  rather than  $R$ , the same argument applies with  $R$  and  $S$  switched.
2. If  $B(R) + B(S)$  is much less than  $M^2$ , we shall have many unused buffers for storing tuples with a common  $Y$ -value, as we suggested in Example 15.8.
3. If all else fails, we can use a nested-loop join on just the tuples with a common  $Y$ -value, using extra disk I/O's but getting the job done correctly. This option was discussed in Section 15.4.5.

#### 15.4.8 Summary of Sort-Based Algorithms

In Fig. 15.11 is a table of the analysis of the algorithms we have discussed in Section 15.4. As discussed in Sections 15.4.5 and 15.4.7, modifications to the time and memory requirements are necessary if we join two relations that have many tuples with the same value in the join attribute(s).

Operators	Approximate $M$ required	Disk I/O	Section
$\gamma, \delta$	$\sqrt{B}$	$3B$	15.4.1, 15.4.2
$\cup, \cap, -$	$\sqrt{B(R) + B(S)}$	$3(B(R) + B(S))$	15.4.3, 15.4.4
$\bowtie$	$\sqrt{\max(B(R), B(S))}$	$5(B(R) + B(S))$	15.4.5
$\bowtie$	$\sqrt{B(R) + B(S)}$	$3(B(R) + B(S))$	15.4.7

Figure 15.11: Main memory and disk I/O requirements for sort-based algorithms

### 15.4.9 Exercises for Section 15.4

**Exercise 15.4.1:** Using the assumptions of Example 15.5 (two tuples per block, etc.),

- a) Show the behavior of the two-pass duplicate-elimination algorithm on the sequence of thirty one-component tuples in which the sequence 0, 1, 2, 3, 4 repeats six times.
- b) Show the behavior of the two-pass grouping algorithm computing the relation  $\gamma_{a, AVG(b)}(R)$ . Relation  $R(a, b)$  consists of the thirty tuples  $t_0$  through  $t_{29}$ , and the tuple  $t_i$  has  $i$  modulo 5 as its grouping component  $a$ , and  $i$  as its second component  $b$ .

**Exercise 15.4.2:** For each of the operations below, write an iterator that uses the algorithm described in this section.

- \* a) Distinct ( $\delta$ ).
- b) Grouping ( $\gamma_L$ ).
- \* c) Set intersection.
- d) Bag difference.
- e) Natural join.

**Exercise 15.4.3:** If  $B(R) = B(S) = 10,000$  and  $M = 1000$ , what are the disk I/O requirements of:

- a) Set union.
- \* b) Simple sort-join.
- c) The more efficient sort-join of Section 15.4.7.

**! Exercise 15.4.4:** Suppose that the second pass of an algorithm described in this section does not need all  $M$  buffers, because there are fewer than  $M$  sublists. How might we save disk I/O's by using the extra buffers?

**! Exercise 15.4.5:** In Example 15.7 we discussed the join of two relations  $R$  and  $S$ , with 1000 and 500 blocks, respectively, and  $M = 101$ . However, we pointed out that there would be additional disk I/O's if there were so many tuples with a given value that neither relation's tuples could fit in main memory. Calculate the total number of disk I/O's needed if:

- \* a) There are only two  $Y$ -values, each appearing in half the tuples of  $R$  and half the tuples of  $S$  (recall  $Y$  is the join attribute or attributes).
- b) There are five  $Y$ -values, each equally likely in each relation.

- c) There are 10  $Y$ -values, each equally likely in each relation.

**! Exercise 15.4.6:** Repeat Exercise 15.4.5 for the more efficient sort-join of Section 15.4.7.

**Exercise 15.4.7:** How much memory do we need to use a two-pass, sort-based algorithm for relations of 10,000 blocks each, if the operation is:

- \* a)  $\delta$ .
- b)  $\gamma$ .
- c) A binary operation such as join or union.

**Exercise 15.4.8:** Describe a two-pass, sort-based algorithm for each of the join-like operators of Exercise 15.2.4.

**! Exercise 15.4.9:** Suppose records could be larger than blocks, i.e., we could have spanned records. How would the memory requirements of two-pass, sort-based algorithms change?

**!! Exercise 15.4.10:** Sometimes, it is possible to save some disk I/O's if we leave the last sublist in memory. It may even make sense to use sublists of fewer than  $M$  blocks to take advantage of this effect. How many disk I/O's can be saved this way?

**!! Exercise 15.4.11:** OQL allows grouping of objects according to arbitrary, user-specified functions of the objects. For example, one could group tuples according to the sum of two attributes. How would we perform a sort-based grouping operation of this type on a set of objects?

## 15.5 Two-Pass Algorithms Based on Hashing

There is a family of hash-based algorithms that attack the same problems as in Section 15.4. The essential idea behind all these algorithms is as follows. If the data is too big to store in main-memory buffers, hash all the tuples of the argument or arguments using an appropriate hash key. For all the common operations, there is a way to select the hash key so all the tuples that need to be considered together when we perform the operation have the same hash value.

We then perform the operation by working on one bucket at a time (or on a pair of buckets with the same hash value, in the case of a binary operation). In effect, we have reduced the size of the operand(s) by a factor equal to the number of buckets. If there are  $M$  buffers available, we can pick  $M$  as the number of buckets, thus gaining a factor of  $M$  in the size of the relations we can handle. Notice that the sort-based algorithms of Section 15.4 also gain a factor of  $M$  by preprocessing, although the sorting and hashing approaches achieve their similar gains by rather different means.

### 15.5.1 Partitioning Relations by Hashing

To begin, let us review the way we would take a relation  $R$  and, using  $M$  buffers, partition  $R$  into  $M - 1$  buckets of roughly equal size. We shall assume that  $h$  is the hash function, and that  $h$  takes complete tuples of  $R$  as its argument (i.e., all attributes of  $R$  are part of the hash key). We associate one buffer with each bucket. The last buffer holds blocks of  $R$ , one at a time. Each tuple  $t$  in the block is hashed to bucket  $h(t)$  and copied to the appropriate buffer. If that buffer is full, we write it out to disk, and initialize another block for the same bucket. At the end, we write out the last block of each bucket if it is not empty. The algorithm is given in more detail in Fig. 15.12. Note that it assumes that tuples, while they may be variable-length, are never too large to fit in an empty buffer.

```

initialize M-1 buckets using M-1 empty buffers;
FOR each block b of relation R DO BEGIN
    read block b into the Mth buffer;
    FOR each tuple t in b DO BEGIN
        IF the buffer for bucket h(t) has no room for t THEN
            BEGIN
                copy the buffer to disk;
                initialize a new empty block in that buffer;
            END;
        copy t to the buffer for bucket h(t);
    END;
END;
FOR each bucket DO
    IF the buffer for this bucket is not empty THEN
        write the buffer to disk;

```

Figure 15.12: Partitioning a relation  $R$  into  $M - 1$  buckets

### 15.5.2 A Hash-Based Algorithm for Duplicate Elimination

We shall now consider the details of hash-based algorithms for the various operations of relational algebra that might need two-pass algorithms. First, consider duplicate elimination, that is, the operation  $\delta(R)$ . We hash  $R$  to  $M - 1$  buckets, as in Fig. 15.12. Note that two copies of the same tuple  $t$  will hash to the same bucket. Thus,  $\delta$  has the essential property we need: we can examine one bucket at a time, perform  $\delta$  on that bucket in isolation, and take as the answer the union of  $\delta(R_i)$ , where  $R_i$  is the portion of  $R$  that hashes to the  $i$ th bucket. The one-pass algorithm of Section 15.2.2 can be used to eliminate

duplicates from each  $R_i$  in turn and write out the resulting unique tuples.

This method will work as long as the individual  $R_i$ 's are sufficiently small to fit in main memory and thus allow a one-pass algorithm. Since we assume the hash function  $h$  partitions  $R$  into equal-sized buckets, each  $R_i$  will be approximately  $B(R)/(M - 1)$  blocks in size. If that number of blocks is no larger than  $M$ , i.e.,  $B(R) \leq M(M - 1)$ , then the two-pass, hash-based algorithm will work. In fact, as we discussed in Section 15.2.2, it is only necessary that the number of distinct tuples in one bucket fit in  $M$  buffers, but we cannot be sure that there are any duplicates at all. Thus, a conservative estimate, with a simple form in which  $M$  and  $M - 1$  are considered the same, is  $B(R) \leq M^2$ , exactly as for the sort-based, two-pass algorithm for  $\delta$ .

The number of disk I/O's is also similar to that of the sort-based algorithm. We read each block of  $R$  once as we hash its tuples, and we write each block of each bucket to disk. We then read each block of each bucket again in the one-pass algorithm that focuses on that bucket. Thus, the total number of disk I/O's is  $3B(R)$ .

### 15.5.3 Hash-Based Grouping and Aggregation

To perform the  $\gamma_L(R)$  operation, we again start by hashing all the tuples of  $R$  to  $M - 1$  buckets. However, in order to make sure that all tuples of the same group wind up in the same bucket, we must choose a hash function that depends only on the grouping attributes of the list  $L$ .

Having partitioned  $R$  into buckets, we can then use the one-pass algorithm for  $\gamma$  from Section 15.2.2 to process each bucket in turn. As we discussed for  $\delta$  in Section 15.5.2, we can process each bucket in main memory provided  $B(R) \leq M^2$ .

However, on the second pass, we only need one record per group as we process each bucket. Thus, even if the size of a bucket is larger than  $M$ , we can handle the bucket in one pass provided the records for all the groups in the bucket take no more than  $M$  buffers. Normally, a group's record will be no larger than a tuple of  $R$ . If so, then a better upper bound on  $B(R)$  is  $M^2$  times the average number of tuples per group.

As a consequence, if there are few groups, then we may actually be able to handle much larger relations  $R$  than is indicated by the  $B(R) \leq M^2$  rule. On the other hand, if  $M$  exceeds the number of groups, then we cannot fill all buckets. Thus, the actual limitation on the size of  $R$  as a function of  $M$  is complex, but  $B(R) \leq M^2$  is a conservative estimate. Finally, we observe that the number of disk I/O's for  $\gamma$ , as for  $\delta$ , is  $3B(R)$ .

### 15.5.4 Hash-Based Union, Intersection, and Difference

When the operation is binary, we must make sure that we use the same hash function to hash tuples of both arguments. For example, to compute  $R \cup_S S$ , we hash both  $R$  and  $S$  to  $M - 1$  buckets each, say  $R_1, R_2, \dots, R_{M-1}$  and

$S_1, S_2, \dots, S_{M-1}$ . We then take the set-union of  $R_i$  with  $S_i$  for all  $i$ , and output the result. Notice that if a tuple  $t$  appears in both  $R$  and  $S$ , then for some  $i$  we shall find  $t$  in both  $R_i$  and  $S_i$ . Thus, when we take the union of these two buckets, we shall output only one copy of  $t$ , and there is no possibility of introducing duplicates into the result. For  $\cup_B$ , the simple bag-union algorithm of Section 15.2.3 is preferable to any other approach for that operation.

To take the intersection or difference of  $R$  and  $S$ , we create the  $2(M - 1)$  buckets exactly as for set-union and apply the appropriate one-pass algorithm to each pair of corresponding buckets. Notice that all these algorithms require  $B(R) + B(S)$  disk I/O's. To this quantity we must add the two disk I/O's per block that are necessary to hash the tuples of the two relations and store the buckets on disk, for a total of  $3(B(R) + B(S))$  disk I/O's.

In order for the algorithms to work, we must be able to take the one-pass union, intersection, or difference of  $R_i$  and  $S_i$ , whose sizes will be approximately  $B(R)/(M - 1)$  and  $B(S)/(M - 1)$ , respectively. Recall that the one-pass algorithms for these operations require that the smaller operand occupies at most  $M - 1$  blocks. Thus, the two-pass, hash-based algorithms require that  $\min(B(R), B(S)) \leq M^2$ , approximately.

### 15.5.5 The Hash-Join Algorithm

To compute  $R(X, Y) \bowtie S(Y, Z)$  using a two-pass, hash-based algorithm, we act almost as for the other binary operations discussed in Section 15.5.4. The only difference is that we must use as the hash key just the join attributes,  $Y$ . Then we can be sure that if tuples of  $R$  and  $S$  join, they will wind up in corresponding buckets  $R_i$  and  $S_i$  for some  $i$ . A one-pass join of all pairs of corresponding buckets completes this algorithm, which we call *hash-join*.<sup>4</sup>

**Example 15.9:** Let us renew our discussion of the two relations  $R$  and  $S$  from Example 15.4, whose sizes were 1000 and 500 blocks, respectively, and for which 101 main-memory buffers are made available. We may hash each relation to 100 buckets, so the average size of a bucket is 10 blocks for  $R$  and 5 blocks for  $S$ . Since the smaller number, 5, is much less than the number of available buffers, we expect to have no trouble performing a one-pass join on each pair of buckets.

The number of disk I/O's is 1500 to read each of  $R$  and  $S$  while hashing into buckets, another 1500 to write all the buckets to disk, and a third 1500 to read each pair of buckets into main memory again while taking the one-pass join of corresponding buckets. Thus, the number of disk I/O's required is 4500, just as for the efficient sort-join of Section 15.4.7.  $\square$

We may generalize Example 15.9 to conclude that:

---

<sup>4</sup>Sometimes, the term “hash-join” is reserved for the variant of the one-pass join algorithm of Section 15.2.3 in which a hash table is used as the main-memory search structure. Then, the two-pass hash-join algorithm described here is called “partition hash-join.”

- Hash join requires  $3(B(R) + B(S))$  disk I/O's to perform its task.
- The two-pass hash-join algorithm will work as long as approximately  $\min(B(R), B(S)) \leq M^2$ .

The argument for the latter point is the same as for the other binary operations: one of each pair of buckets must fit in  $M - 1$  buffers.

### 15.5.6 Saving Some Disk I/O's

If there is more memory available on the first pass than we need to hold one block per bucket, then we have some opportunities to save disk I/O's. One option is to use several blocks for each bucket, and write them out as a group, in consecutive blocks of disk. Strictly speaking, this technique doesn't save disk I/O's, but it makes the I/O's go faster, since we save seek time and rotational latency when we write.

However, there are several tricks that have been used to avoid writing some of the buckets to disk and then reading them again. The most effective of them, called *hybrid hash-join*, works as follows. In general, suppose we decide that to join  $R \bowtie S$ , with  $S$  the smaller relation, we need to create  $k$  buckets, where  $k$  is much less than  $M$ , the available memory. When we hash  $S$ , we can choose to keep  $m$  of the  $k$  buckets entirely in main memory, while keeping only one block for each of the other  $k - m$  buckets. We can manage to do so provided the expected size of the buckets in memory, plus one block for each of the other buckets, does not exceed  $M$ ; that is:

$$\frac{mB(S)}{k} + k - m \leq M \quad (15.1)$$

In explanation, the expected size of a bucket is  $B(S)/k$ , and there are  $m$  buckets in memory.

Now, when we read the tuples of the other relation,  $R$ , to hash that relation into buckets, we keep in memory:

1. The  $m$  buckets of  $S$  that were never written to disk, and
2. One block for each of the  $k - m$  buckets of  $R$  whose corresponding buckets of  $S$  were written to disk.

If a tuple  $t$  of  $R$  hashes to one of the first  $m$  buckets, then we immediately join it with all the tuples of the corresponding  $S$ -bucket, as if this were a one-pass, hash-join. The result of any successful joins is immediately output. It is necessary to organize each of the in-memory buckets of  $S$  into an efficient search structure to facilitate this join, just as for the one-pass hash-join. If  $t$  hashes to one of the buckets whose corresponding  $S$ -bucket is on disk, then  $t$  is sent to the main-memory block for that bucket, and eventually migrates to disk, as for a two-pass, hash-based join.

On the second pass, we join the corresponding buckets of  $R$  and  $S$  as usual. However, there is no need to join the pairs of buckets for which the  $S$ -bucket was left in memory; these buckets have already been joined and their result output.

The savings in disk I/O's is equal to two for every block of the buckets of  $S$  that remain in memory, and their corresponding  $R$ -buckets. Since  $m/k$  of the buckets are in memory, the savings is  $2(m/k)(B(R) + B(S))$ . We must thus ask how to maximize  $m/k$ , subject to the constraint of equation (15.1). The surprising answer is: pick  $m = 1$ , and then make  $k$  as small as possible.

The intuitive justification is that all but  $k - m$  of the main-memory buffers can be used to hold tuples of  $S$  in main memory, and the more of these tuples, the fewer the disk I/O's. Thus, we want to minimize  $k$ , the total number of buckets. We do so by making each bucket about as big as can fit in main memory; that is, buckets are of size  $M$ , and therefore  $k = B(S)/M$ . If that is the case, then there is only room for one bucket in the extra main memory; i.e.,  $m = 1$ .

In fact, we really need to make the buckets slightly smaller than  $B(S)/M$ , or else we shall not quite have room for one full bucket and one block for the other  $k - 1$  buckets in memory at the same time. Assuming, for simplicity, that  $k$  is about  $B(S)/M$  and  $m = 1$ , the savings in disk I/O's is

$$\left(\frac{2M}{B(S)}\right)(B(R) + B(S))$$

and the total cost is

$$\left(3 - \frac{2M}{B(S)}\right)(B(R) + B(S))$$

**Example 15.10:** Consider the problem of Example 15.4, where we had to join relations  $R$  and  $S$ , of 1000 and 500 blocks, respectively, using  $M = 101$ . If we use a hybrid hash-join, then we want  $k$ , the number of buckets, to be about  $500/101$ . Suppose we pick  $k = 5$ . Then the average bucket will have 100 blocks of  $S$ 's tuples. If we try to fit one of these buckets and four extra blocks for the other four buckets, we need 104 blocks of main memory, and we cannot take the chance that the in-memory bucket will overflow memory.

Thus, we are advised to choose  $k = 6$ . Now, when hashing  $S$  on the first pass, we have five buffers for five of the buckets, and we have up to 96 buffers for the in-memory bucket, whose expected size is  $500/6$  or 83. The number of disk I/O's we use for  $S$  on the first pass is thus 500 to read all of  $S$ , and  $500 - 83 = 417$  to write five buckets to disk. When we process  $R$  on the first pass, we need to read all of  $R$  (1000 disk I/O's) and write 5 of its 6 buckets (833 disk I/O's).

On the second pass, we read all the buckets written to disk, or  $417 + 833 = 1250$  additional disk I/O's. The total number of disk I/O's is thus 1500 to read  $R$  and  $S$ , 1250 to write 5/6 of these relations, and another 1250 to read those tuples again, or 4000 disk I/O's. This figure compares with the 4500 disk I/O's needed for the straightforward hash-join or sort-join.  $\square$

### 15.5.7 Summary of Hash-Based Algorithms

Figure 15.13 gives the memory requirements and disk I/O's needed by each of the algorithms discussed in this section. As with other types of algorithms, we should observe that the estimates for  $\gamma$  and  $\delta$  may be conservative, since they really depend on the number of duplicates and groups, respectively, rather than on the number of tuples in the argument relation.

Operators	Approximate $M$ required	Disk I/O	Section
$\gamma, \delta$	$\sqrt{B}$	$3B$	15.5.2, 15.5.3
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$(3 - 2M/B(S))(B(R) + B(S))$	15.5.6

Figure 15.13: Main memory and disk I/O requirements for hash-based algorithms; for binary operations, assume  $B(S) \leq B(R)$

Notice that the requirements for sort-based and the corresponding hash-based algorithms are almost the same. The significant differences between the two approaches are:

1. Hash-based algorithms for binary operations have a size requirement that depends only on the smaller of two arguments rather than on the sum of the argument sizes, as for sort-based algorithms.
2. Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later. The result might be used in another sort-based algorithm later, or it could be the answer to a query that is required to be produced in sorted order.
3. Hash-based algorithms depend on the buckets being of equal size. Since there is generally at least a small variation in size, it is not possible to use buckets that, on average, occupy  $M$  blocks; we must limit them to a somewhat smaller figure. This effect is especially prominent if the number of different hash keys is small, e.g., performing a group-by on a relation with few groups or a join with very few values for the join attributes.
4. In sort-based algorithms, the sorted sublists may be written to consecutive blocks of the disk if we organize the disk properly. Thus, one of the three disk I/O's per block may require little rotational latency or seek time

and therefore may be much faster than the I/O's needed for hash-based algorithms.

5. Moreover, if  $M$  is much larger than the number of sorted sublists, then we may read in several consecutive blocks at a time from a sorted sublist, again saving some latency and seek time.
6. On the other hand, if we can choose the number of buckets to be less than  $M$  in a hash-based algorithm, then we can write out several blocks of a bucket at once. We thus obtain the same benefit on the write step for hashing that the sort-based algorithms have for the second read, as we observed in (5). Similarly, we may be able to organize the disk so that a bucket eventually winds up on consecutive blocks of tracks. If so, buckets can be read with little latency or seek time, just as sorted sublists were observed in (4) to be writable efficiently.

### 15.5.8 Exercises for Section 15.5

**Exercise 15.5.1:** The hybrid-hash-join idea, storing one bucket in main memory, can also be applied to other operations. Show how to save the cost of storing and reading one bucket from each relation when implementing a two-pass, hash-based algorithm for: \*a)  $\delta$  b)  $\gamma$  c)  $\cap_B$  d)  $-_S$ .

**Exercise 15.5.2:** If  $B(S) = B(R) = 10,000$  and  $M = 1000$ , what is the number of disk I/O's required for a hybrid hash join?

**Exercise 15.5.3:** Write iterators that implement the two-pass, hash-based algorithms for a)  $\delta$  b)  $\gamma$  c)  $\cap_B$  d)  $-_S$  e)  $\bowtie$ .

**\*! Exercise 15.5.4:** Suppose we are performing a two-pass, hash-based grouping operation on a relation  $R$  of the appropriate size; i.e.,  $B(R) \leq M^2$ . However, there are so few groups, that some groups are larger than  $M$ ; i.e., they will not fit in main memory at once. What modifications, if any, need to be made to the algorithm given here?

**! Exercise 15.5.5:** Suppose that we are using a disk where the time to move the head to a block is 100 milliseconds, and it takes 1/2 millisecond to read one block. Therefore, it takes  $k/2$  milliseconds to read  $k$  consecutive blocks, once the head is positioned. Suppose we want to compute a two-pass hash-join  $R \bowtie S$ , where  $B(R) = 1000$ ,  $B(S) = 500$ , and  $M = 101$ . To speed up the join, we want to use as few buckets as possible (assuming tuples distribute evenly among buckets), and read and write as many blocks as we can to consecutive positions on disk. Counting 100.5 milliseconds for a random disk I/O and  $100 + k/2$  milliseconds for reading or writing  $k$  consecutive blocks from or to disk:

- a) How much time does the disk I/O take?

- b) How much time does the disk I/O take if we use a hybrid hash-join as described in Example 15.10?
- c) How much time does a sort-based join take under the same conditions, assuming we write sorted sublists to consecutive blocks of disk?

## 15.6 Index-Based Algorithms

The existence of an index on one or more attributes of a relation makes available some algorithms that would not be feasible without the index. Index-based algorithms are especially useful for the selection operator, but algorithms for join and other binary operators also use indexes to very good advantage. In this section, we shall introduce these algorithms. We also continue with the discussion of the index-scan operator for accessing a stored table with an index that we began in Section 15.1.1. To appreciate many of the issues, we first need to digress and consider “clustering” indexes.

### 15.6.1 Clustering and Nonclustering Indexes

Recall from Section 15.1.3 that a relation is “clustered” if its tuples are packed into roughly as few blocks as can possibly hold those tuples. All the analyses we have done so far assume that relations are clustered.

We may also speak of *clustering indexes*, which are indexes on an attribute or attributes such that all the tuples with a fixed value for the search key of this index appear on roughly as few blocks as can hold them. Note that a relation that isn’t clustered cannot have a clustering index,<sup>5</sup> but even a clustered relation can have nonclustering indexes.

**Example 15.11:** A relation  $R(a, b)$  that is sorted on attribute  $a$  and stored in that order, packed into blocks, is surely clustered. An index on  $a$  is a clustering index, since for a given  $a$ -value  $a_1$ , all the tuples with that value for  $a$  are consecutive. They thus appear packed into blocks, except possibly for the first and last blocks that contain  $a$ -value  $a_1$ , as suggested in Fig. 15.14. However, an index on  $b$  is unlikely to be clustering, since the tuples with a fixed  $b$ -value will be spread all over the file unless the values of  $a$  and  $b$  are very closely correlated.  $\square$

---

<sup>5</sup>Technically, if the index is on a key for the relation, so only one tuple with a given value in the index key exists, then the index is always “clustering,” even if the relation is not clustered. However, if there is only one tuple per index-key value, then there is no advantage from clustering, and the performance measure for such an index is the same as if it were considered nonclustering.

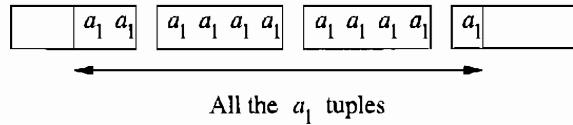


Figure 15.14: A clustering index has all tuples with a fixed value packed into (close to) the minimum possible number of blocks

### 15.6.2 Index-Based Selection

In Section 15.1.1 we discussed implementing a selection  $\sigma_C(R)$  by reading all the tuples of relation  $R$ , seeing which meet the condition  $C$ , and outputting those that do. If there are no indexes on  $R$ , then that is the best we can do; the number of disk I/O's used by the operation is  $B(R)$ , or even  $T(R)$ , the number of tuples of  $R$ , should  $R$  not be a clustered relation.<sup>6</sup> However, suppose that the condition  $C$  is of the form  $a = v$ , where  $a$  is an attribute for which an index exists, and  $v$  is a value. Then one can search the index with value  $v$  and get pointers to exactly those tuples of  $R$  that have  $a$ -value  $v$ . These tuples constitute the result of  $\sigma_{a=v}(R)$ , so all we have to do is retrieve them.

If the index on  $R.a$  is clustering, then the number of disk I/O's to retrieve the set  $\sigma_{a=v}(R)$  will average  $B(R)/V(R, a)$ . The actual number may be somewhat higher, because:

1. Often, the index is not kept entirely in main memory, and therefore some disk I/O's are needed to support the index lookup.
2. Even though all the tuples with  $a = v$  might fit in  $b$  blocks, they could be spread over  $b + 1$  blocks because they don't start at the beginning of a block.
3. Although the index is clustering, the tuples with  $a = v$  may be spread over several extra blocks. Two reasons why that situation might occur are:
  - (a) We might not pack blocks of  $R$  as tightly as possible because we want to leave room for growth of  $R$ , as discussed in Section 13.1.6.
  - (b)  $R$  might be stored with some other tuples that do not belong to  $R$ , say in a clustered-file organization.

Moreover, we of course must round up if the ratio  $B(R)/V(R, a)$  is not an integer. Most significant is that should  $a$  be a key for  $R$ , then  $V(R, a) = T(R)$ , which is presumably much bigger than  $B(R)$ , yet we surely require one disk I/O to retrieve the tuple with key value  $v$ , plus whatever disk I/O's are needed to access the index.

---

<sup>6</sup>Recall from Section 15.1.3 the notation we developed:  $T(R)$  for the number of tuples in  $R$ ,  $B(R)$  for the number of blocks in which  $R$  fits, and  $V(R, L)$  for the number of distinct tuples in  $\pi_L(R)$ .

### Notions of Clustering

We have seen three different, although related, concepts called “clustering” or “clustered.”

1. In Section 13.2.2 we spoke of the “clustered-file organization,” where tuples of one relation  $R$  are placed with a tuple of some other relation  $S$  with which they share a common value; the example was grouping movie tuples with the tuple of the studio that made the movie.
2. In Section 15.1.3 we spoke of a “clustered relation,” meaning that the tuples of the relation are stored in blocks that are exclusively, or at least predominantly, devoted to storing that relation.
3. Here, we have introduced the notion of a clustering index — an index in which the tuples having a given value of the search key appear in blocks that are largely devoted to storing tuples with that search-key value. Typically, the tuples with a fixed value will be stored consecutively, and only the first and last blocks with tuples of that value will also have tuples of another search-key value.

The clustered-file organization is one example of a way to have a clustered relation that is not packed into blocks which are exclusively its own. Suppose that one tuple of the relation  $S$  is associated with many  $R$ -tuples in a clustered file. Then, while the tuples of  $R$  are not packed in blocks exclusively devoted to  $R$ , these blocks are “predominantly” devoted to  $R$ , and we call  $R$  clustered. On the other hand,  $S$  will typically *not* be a clustered relation, since its tuples are usually on blocks devoted predominantly to  $R$ -tuples rather than  $S$ -tuples.

Now, let us consider what happens when the index on  $R.a$  is nonclustering. To a first approximation, each tuple we retrieve will be on a different block, and we must access  $T(R)/V(R,a)$  tuples. Thus,  $T(R)/V(R,a)$  is an estimate of the number of disk I/O’s we need. The number could be higher because we may also need to read some index blocks from disk; it could be lower because fortuitously some retrieved tuples appear on the same block, and that block remains buffered in memory.

**Example 15.12:** Suppose  $B(R) = 1000$ , and  $T(R) = 20,000$ . That is,  $R$  has 20,000 tuples that are packed 20 to a block. Let  $a$  be one of the attributes of  $R$ , suppose there is an index on  $a$ , and consider the operation  $\sigma_{a=0}(R)$ . Here are some possible situations and the worst-case number of disk I/O’s required. We shall ignore the cost of accessing the index blocks in all cases.

1. If  $R$  is clustered, but we do not use the index, then the cost is 1000 disk

I/O's. That is, we must retrieve every block of  $R$ .

2. If  $R$  is not clustered and we do not use the index, then the cost is 20,000 disk I/O's.
3. If  $V(R, a) = 100$  and the index is clustering, then the index-based algorithm uses  $1000/100 = 10$  disk I/O's.
4. If  $V(R, a) = 10$  and the index is nonclustering, then the index-based algorithm uses  $20,000/10 = 2000$  disk I/O's. Notice that this cost is higher than scanning the entire relation  $R$ , if  $R$  is clustered but the index is not.
5. If  $V(R, a) = 20,000$ , i.e.,  $a$  is a key, then the index-based algorithm takes 1 disk I/O plus whatever is needed to access the index, regardless of whether the index is clustering or not.

□

Index-scan as an access method can help in several other kinds of selection operations.

- a) An index such as a B-tree lets us access the search-key values in a given range efficiently. If such an index on attribute  $a$  of relation  $R$  exists, then we can use the index to retrieve just the tuples of  $R$  in the desired range for selections such as  $\sigma_{a \geq 10}(R)$ , or even  $\sigma_{a \geq 10} \text{ AND } a \leq 20(R)$ .
- b) A selection with a complex condition  $C$  can sometimes be implemented by an index-scan followed by another selection on only those tuples retrieved by the index-scan. If  $C$  is of the form  $a = v \text{ AND } C'$ , where  $C'$  is any condition, then we can split the selection into a cascade of two selections, the first checking only for  $a = v$ , and the second checking condition  $C'$ . The first is a candidate for use of the index-scan operator. This splitting of a selection operation is one of many improvements that a query optimizer may make to a logical query plan; it is discussed particularly in Section 16.7.1.

### 15.6.3 Joining by Using an Index

All the binary operations we have considered, and the unary full-relation operations of  $\gamma$  and  $\delta$  as well, can use certain indexes profitably. We shall leave most of these algorithms as exercises, while we focus on the matter of joins. In particular, let us examine the natural join  $R(X, Y) \bowtie S(Y, Z)$ ; recall that  $X$ ,  $Y$ , and  $Z$  can stand for sets of attributes, although it is adequate to think of them as single attributes.

For our first index-based join algorithm, suppose that  $S$  has an index on the attribute(s)  $Y$ . Then one way to compute the join is to examine each block of  $R$ , and within each block consider each tuple  $t$ . Let  $t_Y$  be the component or

components of  $t$  corresponding to the attribute(s)  $Y$ . Use the index to find all those tuples of  $S$  that have  $t_Y$  in their  $Y$ -component(s). These are exactly the tuples of  $S$  that join with tuple  $t$  of  $R$ , so we output the join of each of these tuples with  $t$ .

The number of disk I/O's depends on several factors. First, assuming  $R$  is clustered, we shall have to read  $B(R)$  blocks to get all the tuples of  $R$ . If  $R$  is not clustered, then up to  $T(R)$  disk I/O's may be required.

For each tuple  $t$  of  $R$  we must read an average of  $T(S)/V(S,Y)$  tuples of  $S$ . If  $S$  has a nonclustered index on  $Y$ , then the number of disk I/O's required to read  $S$  is  $T(R)T(S)/V(S,Y)$ , but if the index is clustered, then only  $T(R)B(S)/V(S,Y)$  disk I/O's suffice.<sup>7</sup> In either case, we may have to add a few disk I/O's per  $Y$ -value, to account for the reading of the index itself.

Regardless of whether or not  $R$  is clustered, the cost of accessing tuples of  $S$  dominates. Ignoring the cost of reading  $R$ , we shall take  $T(R)T(S)/V(S,Y)$  or  $T(R)(\max(1, B(S)/V(S,Y)))$  as the cost of this join method, for the cases of nonclustered and clustered indexes on  $S$ , respectively.

**Example 15.13:** Let us consider our running example, relations  $R(X, Y)$  and  $S(Y, Z)$  covering 1000 and 500 blocks, respectively. Assume ten tuples of either relation fit on one block, so  $T(R) = 10,000$  and  $T(S) = 5000$ . Also, assume  $V(S, Y) = 100$ ; i.e., there are 100 different values of  $Y$  among the tuples of  $S$ .

Suppose that  $R$  is clustered, and there is a clustering index on  $Y$  for  $S$ . Then the approximate number of disk I/O's, excluding what is needed to access the index itself, is 1000 to read the blocks of  $R$  (neglected in the formulas above) plus  $10,000 \times 500 / 100 = 50,000$  disk I/O's. This number is considerably above the cost of other methods for the same data discussed previously. If either  $R$  or the index on  $S$  is not clustered, then the cost is even higher.  $\square$

While Example 15.13 makes it look as if an index-join is a very bad idea, there are other situations where the join  $R \bowtie S$  by this method makes much more sense. Most common is the case where  $R$  is very small compared with  $S$ , and  $V(S, Y)$  is large. We discuss in Exercise 15.6.5 a typical query in which selection before a join makes  $R$  tiny. In that case, most of  $S$  will never be examined by this algorithm, since most  $Y$ -values don't appear in  $R$  at all. However, both sort- and hash-based join methods will examine every tuple of  $S$  at least once.

#### 15.6.4 Joins Using a Sorted Index

When the index is a B-tree, or any other structure from which we easily can extract the tuples of a relation in sorted order, we have a number of other opportunities to use the index. Perhaps the simplest is when we want to compute  $R(X, Y) \bowtie S(Y, Z)$ , and we have such an index on  $Y$  for either  $R$  or  $S$ . We

---

<sup>7</sup>But remember that  $B(S)/V(S, Y)$  must be replaced by 1 if it is less, as discussed in Section 15.6.2.

can then perform an ordinary sort-join, but we do not have to perform the intermediate step of sorting one of the relations on  $Y$ .

As an extreme case, if we have sorting indexes on  $Y$  for both  $R$  and  $S$ , then we need to perform only the final step of the simple sort-based join of Section 15.4.5. This method is sometimes called *zig-zag join*, because we jump back and forth between the indexes finding  $Y$ -values that they share in common. Notice that tuples from  $R$  with a  $Y$ -value that does not appear in  $S$  need never be retrieved, and similarly, tuples of  $S$  whose  $Y$ -value does not appear in  $R$  need not be retrieved.

**Example 15.14:** Suppose that we have relations  $R(X, Y)$  and  $S(Y, Z)$  with indexes on  $Y$  for both relations. In a tiny example, let the search keys ( $Y$ -values) for the tuples of  $R$  be in order 1, 3, 4, 4, 4, 5, 6, and let the search key values for  $S$  be 2, 2, 4, 4, 6, 7. We start with the first keys of  $R$  and  $S$ , which are 1 and 2, respectively. Since  $1 < 2$ , we skip the first key of  $R$  and look at the second key, 3. Now, the current key of  $S$  is less than the current key of  $R$ , so we skip the two 2's of  $S$  to reach 4.

At this point, the key 3 of  $R$  is less than the key of  $S$ , so we skip the key of  $R$ . Now, both current keys are 4. We follow the pointers associated with all the keys 4 from both relations, retrieve the corresponding tuples, and join them. Notice that until we met the common key 4, no tuples of the relation were retrieved.

Having dispensed with the 4's, we go to key 5 of  $R$  and key 6 of  $S$ . Since  $5 < 6$ , we skip to the next key of  $R$ . Now the keys are both 6, so we retrieve the corresponding tuples and join them. Since  $R$  is now exhausted, we know there are no more pairs of tuples from the two relations that join.  $\square$

If the indexes are B-trees, then we can scan the leaves of the two B-trees in order from the left, using the pointers from leaf to leaf that are built into the structure, as suggested in Fig. 15.15. If  $R$  and  $S$  are clustered, then retrieval of all the tuples with a given key will result in a number of disk I/O's proportional to the fractions of these two relations read. Note that in extreme cases, where there are so many tuples from  $R$  and  $S$  that neither fits in the available main memory, we shall have to use a fixup like that discussed in Section 15.4.5. However, in typical cases, the step of joining all tuples with a common  $Y$ -value can be carried out with only as many disk I/O's as it takes to read them.

**Example 15.15:** Let us continue with Example 15.13, to see how joins using a combination of sorting and indexing would typically perform on this data. First, assume that there is an index on  $Y$  for  $S$  that allows us to retrieve the tuples of  $S$  sorted by  $Y$ . We shall, in this example, also assume both relations and the index are clustered. For the moment, we assume there is no index on  $R$ .

Assuming 101 available blocks of main memory, we may use them to create 10 sorted sublists for the 1000-block relation  $R$ . The number of disk I/O's is 2000 to read and write all of  $R$ . We next use 11 blocks of memory — 10 for

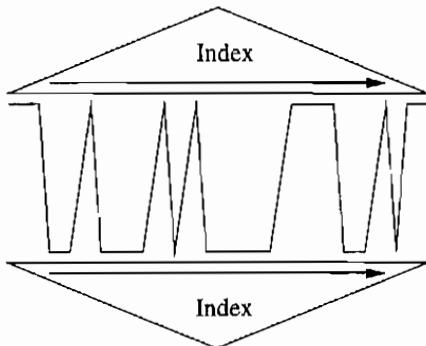


Figure 15.15: A zig-zag join using two indexes

the sublists of  $R$  and one for a block of  $S$ 's tuples, retrieved via the index. We neglect disk I/O's and memory buffers needed to manipulate the index, but if the index is a B-tree, these numbers will be small anyway. In this second pass, we read all the tuples of  $R$  and  $S$ , using a total of 1500 disk I/O's, plus the small amount needed for reading the index blocks once each. We thus estimate the total number of disk I/O's at 3500, which is less than that for other methods considered so far.

Now, assume that both  $R$  and  $S$  have indexes on  $Y$ . Then there is no need to sort either relation. We use just 1500 disk I/O's to read the blocks of  $R$  and  $S$  through their indexes. In fact, if we determine from the indexes alone that a large fraction of  $R$  or  $S$  cannot match tuples of the other relation, then the total cost could be considerably less than 1500 disk I/O's. However, in any event we should add the small number of disk I/O's needed to read the indexes themselves.  $\square$

### 15.6.5 Exercises for Section 15.6

**Exercise 15.6.1:** Suppose there is an index on attribute  $R.a$ . Describe how this index could be used to improve the execution of the following operations. Under what circumstances would the index-based algorithm be more efficient than sort- or hash-based algorithms?

- \* a)  $R \cup_S S$  (assume that  $R$  and  $S$  have no duplicates, although they may have tuples in common).
- b)  $R \cap_S S$  (again, with  $R$  and  $S$  sets).
- c)  $\delta(R)$ .

**Exercise 15.6.2:** Suppose  $B(R) = 10,000$  and  $T(R) = 500,000$ . Let there be an index on  $R.a$ , and let  $V(R, a) = k$  for some number  $k$ . Give the cost of  $\sigma_{a=0}(R)$ , as a function of  $k$ , under the following circumstances. You may neglect disk I/O's needed to access the index itself.

- \* a) The index is clustering.
- b) The index is not clustering.
- c)  $R$  is clustered, and the index is not used.

**Exercise 15.6.3:** Repeat Exercise 15.6.2 if the operation is the range query  $\sigma_{C \leq a \text{ AND } a \leq D}(R)$ . You may assume that  $C$  and  $D$  are constants such that  $k/10$  of the values are in the range.

**! Exercise 15.6.4:** If  $R$  is clustered, but the index on  $R.a$  is *not* clustering, then depending on  $k$  we may prefer to implement a query by performing a table-scan of  $R$  or using the index. For what values of  $k$  would we prefer to use the index if the relation and query are as in:

- a) Exercise 15.6.2.
- b) Exercise 15.6.3.

**\* Exercise 15.6.5:** Consider the SQL query:

```
SELECT birthdate
  FROM StarsIn, MovieStar
 WHERE movieTitle = 'King Kong' AND starName = name;
```

This query uses the “movie” relations:

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
```

If we translate it to relational algebra, the heart is an equijoin between

$$\sigma_{\text{movieTitle} = 'King Kong'}(\text{StarsIn})$$

and **MovieStar**, which can be implemented much as a natural join  $R \bowtie S$ . Since there were only two movies named “King Kong,”  $T(R)$  is very small. Suppose that  $S$ , the relation **MovieStar**, has an index on **name**. Compare the cost of an index-join for this  $R \bowtie S$  with the cost of a sort- or hash-based join.

**! Exercise 15.6.6:** In Example 15.15 we discussed the disk-I/O cost of a join  $R \bowtie S$  in which one or both of  $R$  and  $S$  had sorting indexes on the join attribute(s). However, the methods described in that example can fail if there are too many tuples with the same value in the join attribute(s). What are the limits (in number of blocks occupied by tuples with the same value) under which the methods described will not need to do additional disk I/O’s?

## 15.7 Buffer Management

We have assumed that operators on relations have available some number  $M$  of main-memory buffers that they can use to store needed data. In practice, these buffers are rarely allocated in advance to the operator, and the value of  $M$  may vary depending on system conditions. The central task of making main-memory buffers available to processes, such as queries, that act on the database is given to the *buffer manager*. It is the responsibility of the buffer manager to allow processes to get the memory they need, while minimizing the delay and unsatisfiable requests. The role of the buffer manager is illustrated in Fig. 15.16.

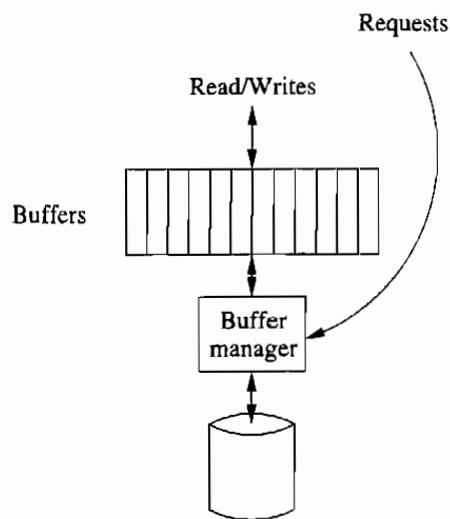


Figure 15.16: The buffer manager responds to requests for main-memory access to disk blocks

### 15.7.1 Buffer Management Architecture

There are two broad architectures for a buffer manager:

1. The buffer manager controls main memory directly, as in many relational DBMS's, or
2. The buffer manager allocates buffers in virtual memory, allowing the operating system to decide which buffers are actually in main memory at any time and which are in the “swap space” on disk that the operating system manages. Many “main-memory” DBMS's and “object-oriented” DBMS's operate this way.

Whichever approach a DBMS uses, the same problem arises: the buffer manager should limit the number of buffers in use so they fit in the available

### Memory Management for Query Processing

We are assuming that the buffer manager allocates to an operator  $M$  main-memory buffers, where the value for  $M$  depends on system conditions (including other operators and queries underway), and may vary dynamically. Once an operator has  $M$  buffers, it may use some of them for bringing in disk pages, others for index pages, and still others for sort runs or hash tables. In some DBMS's, memory is not allocated from a single pool, but rather there are separate pools of memory — with separate buffer managers — for different purposes. For example, an operator might be allocated  $D$  buffers from a pool to hold pages brought in from disk,  $S$  buffers from a separate memory area allocated for sorting, and  $H$  buffers to build a hash table. This approach offers more opportunities for system configuration and “tuning,” but may not make the best global use of memory.

main memory. When the buffer manager controls main memory directly, and requests exceed available space, it has to select a buffer to empty, by returning its contents to disk. If the buffered block has not been changed, then it may simply be erased from main memory, but if the block has changed it must be written back to its place on the disk. When the buffer manager allocates space in virtual memory, it has the option to allocate more buffers than can fit in main memory. However, if all these buffers are really in use, then there will be “thrashing,” a common operating-system problem, where many blocks are moved in and out of the disk’s swap space. In this situation, the system spends most of its time swapping blocks, while very little useful work gets done.

Normally, the number of buffers is a parameter set when the DBMS is initialized. We would expect that this number is set so that the buffers occupy the available main memory, regardless of whether the buffers are allocated in main or virtual memory. In what follows, we shall not concern ourselves with which mode of buffering is used, and simply assume that there is a fixed-size *buffer pool*, a set of buffers available to queries and other database actions.

#### 15.7.2 Buffer Management Strategies

The critical choice that the buffer manager must make is what block to throw out of the buffer pool when a buffer is needed for a newly requested block. The *buffer-replacement strategies* in common use may be familiar to you from other applications of scheduling policies, such as in operating systems. These include:

### Least-Recently Used (LRU)

The LRU rule is to throw out the block that has not been read or written for the longest time. This method requires that the buffer manager maintain a table indicating the last time the block in each buffer was accessed. It also requires that each database access make an entry in this table, so there is significant effort in maintaining this information. However, LRU is an effective strategy; intuitively, buffers that have not been used for a long time are less likely to be accessed sooner than those that have been accessed recently.

### First-In-First-Out (FIFO)

When a buffer is needed, under the FIFO policy the buffer that has been occupied the longest by the same block is emptied and used for the new block. In this approach, the buffer manager needs to know only the time at which the block currently occupying a buffer was loaded into that buffer. An entry into a table can thus be made when the block is read from disk, and there is no need to modify the table when the block is accessed. FIFO requires less maintenance than LRU, but it can make more mistakes. A block that is used repeatedly, say the root block of a B-tree index, will eventually become the oldest block in a buffer. It will be written back to disk, only to be reread shortly thereafter into another buffer.

### The “Clock” Algorithm (“Second Chance”)

This algorithm is a commonly implemented, efficient approximation to LRU. Think of the buffers as arranged in a circle, as suggested by Fig. 15.17. A “hand” points to one of the buffers, and will rotate clockwise if it needs to find a buffer in which to place a disk block. Each buffer has an associated “flag,” which is either 0 or 1. Buffers with a 0 flag are vulnerable to having their contents sent back to disk; buffers with a 1 are not. When a block is read into a buffer, its flag is set to 1. Likewise, when the contents of a buffer is accessed, its flag is set to 1.

When the buffer manager needs a buffer for a new block, it looks for the first 0 it can find, rotating clockwise. If it passes 1's, it sets them to 0. Thus, a block is only thrown out of its buffer if it remains unaccessed for the time it takes the hand to make a complete rotation to set its flag to 0 and then make another complete rotation to find the buffer with its 0 unchanged. For instance, in Fig. 15.17, the hand will set to 0 the 1 in the buffer to its left, and then move clockwise to find the buffer with 0, whose block it will replace and whose flag it will set to 1.

### System Control

The query processor or other components of a DBMS can give advice to the buffer manager in order to avoid some of the mistakes that would occur with

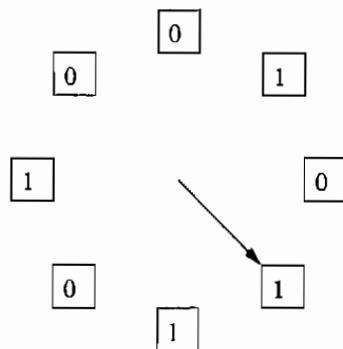


Figure 15.17: The clock algorithm visits buffers in a round-robin fashion and replaces  $01 \dots 1$  with  $10 \dots 0$

### More Tricks Using the Clock Algorithm

The “clock” algorithm for choosing buffers to free is not limited to the scheme described in Section 15.7.2, where flags had values 0 and 1. For instance, one can start an important page with a number higher than 1 as its flag, and decrement the flag by 1 each time the “hand” passes that page. In fact, one can incorporate the concept of pinning blocks by giving the pinned block an infinite value for its flag, and then having the system release the pin at the appropriate time by setting the flag to 0.

a strict policy such as LRU, FIFO, or Clock. Recall from Section 12.3.5 that there are sometimes technical reasons why a block in main memory can *not* be moved to disk without first modifying certain other blocks that point to it. These blocks are called “pinned,” and any buffer manager has to modify its buffer-replacement strategy to avoid expelling pinned blocks. This fact gives us the opportunity to force other blocks to remain in main memory by declaring them “pinned,” even if there is no technical reason why they could not be written to disk. For example, a cure for the problem with FIFO mentioned above regarding the root of a B-tree is to “pin” the root, forcing it to remain in memory at all times. Similarly, for an algorithm like a one-pass hash-join, the query processor may “pin” the blocks of the smaller relation in order to assure that it will remain in main memory during the entire time.

### 15.7.3 The Relationship Between Physical Operator Selection and Buffer Management

The query optimizer will eventually select a set of physical operators that will be used to execute a given query. This selection of operators may assume that a certain number of buffers  $M$  is available for execution of each of these operators.

However, as we have seen, the buffer manager may not be willing or able to guarantee the availability of these  $M$  buffers when the query is executed. There are thus two related questions to ask about the physical operators:

1. Can the algorithm adapt to changes in the value of  $M$ , the number of main-memory buffers available?
2. When the expected  $M$  buffers are not available, and some blocks that are expected to be in memory have actually been moved to disk by the buffer manager, how does the buffer-replacement strategy used by the buffer manager impact the number of additional I/O's that must be performed?

**Example 15.16:** As an example of the issues, let us consider the block-based nested-loop join of Fig. 15.8. The basic algorithm does not really depend on the value of  $M$ , although its performance depends on  $M$ . Thus, it is sufficient to find out what  $M$  is just before execution begins.

It is even possible that  $M$  will change at different iterations of the outer loop. That is, each time we load main memory with a portion of the relation  $S$  (the relation of the outer loop), we can use all but one of the buffers available at that time; the remaining buffer is reserved for a block of  $R$ , the relation of the inner loop. Thus, the number of times we go around the outer loop depends on the average number of buffers available at each iteration. However, as long as  $M$  buffers are available *on average*, then the cost analysis of Section 15.3.4 will hold. In the extreme, we might have the good fortune to find that at the first iteration, enough buffers are available to hold all of  $S$ , in which case nested-loop join gracefully becomes the one-pass join of Section 15.2.3.

If we pin the  $M - 1$  blocks we use for  $S$  on one iteration of the outer loop, then we shall not lose their buffers during the round. On the other hand, more buffers may become available during that iteration. These buffers allow more than one block of  $R$  to be kept in memory at the same time, but unless we are careful, the extra buffers will not improve the running time of the nested-loop join.

For instance, suppose that we use an LRU buffer-replacement strategy, and there are  $k$  buffers available to hold blocks of  $R$ . As we read each block of  $R$ , in order, the blocks that remain in buffers at the end of this iteration of the outer loop will be the last  $k$  blocks of  $R$ . We next reload the  $M - 1$  buffers for  $S$  with new blocks of  $S$  and start reading the blocks of  $R$  again, in the next iteration of the outer loop. However, if we start from the beginning of  $R$  again, then the  $k$  buffers for  $R$  will need to be replaced, and we do not save disk I/O's just because  $k > 1$ .

A better implementation of nested-loop join, when an LRU buffer-replacement strategy is used, visits the blocks of  $R$  in an order that alternates: first-to-last and then last-to-first (called *rocking*). In that way, if there are  $k$  buffers available to  $R$ , we save  $k$  disk I/O's on each iteration of the outer loop except the first. That is, the second and subsequent iterations require only  $B(R) - k$

disk I/O's for  $R$ . Notice that even if  $k = 1$  (i.e., no *extra* buffers are available to  $R$ ), we save one disk I/O per iteration.  $\square$

Other algorithms also are impacted by the fact that  $M$  can vary and by the buffer-replacement strategy used by the buffer manager. Here are some useful observations.

- If we use a sort-based algorithm for some operator, then it is possible to adapt to changes in  $M$ . If  $M$  shrinks, we can change the size of a sublist, since the sort-based algorithms we discussed do not depend on the sublists being the same size. The major limitation is that as  $M$  shrinks, we could be forced to create so many sublists that we cannot then allocate a buffer for each sublist in the merging process.
- The main-memory sorting of sublists can be performed by a number of different algorithms. Since algorithms like merge-sort and quicksort are recursive, most of the time is spent on rather small regions of memory. Thus, either LRU or FIFO will perform well for this part of a sort-based algorithm.
- If the algorithm is hash-based, we can reduce the number of buckets if  $M$  shrinks, as long as the buckets do not then become so large that they do not fit in allotted main memory. However, unlike sort-based algorithms, we cannot respond to changes in  $M$  while the algorithm runs. Rather, once the number of buckets is chosen, it remains fixed throughout the first pass, and if buffers become unavailable, the blocks belonging to some of the buckets will have to be swapped out.

#### 15.7.4 Exercises for Section 15.7

**Exercise 15.7.1:** Suppose that we wish to execute a join  $R \bowtie S$ , and the available memory will vary between  $M$  and  $M/2$ . In terms of  $M$ ,  $B(R)$ , and  $B(S)$ , give the conditions under which we can guarantee that the following algorithms can be executed:

- \* a) A one-pass join.
- \* b) A two-pass, hash-based join.
- c) A two-pass, sort-based join.

! **Exercise 15.7.2:** How would the number of disk I/O's taken by a nested-loop join improve if extra buffers became available and the buffer-replacement policy were:

- a) First-in-first-out.
- b) The clock algorithm.

**!! Exercise 15.7.3:** In Example 15.16, we suggested that it was possible to take advantage of extra buffers becoming available during the join by keeping more than one block of  $R$  buffered and visiting the blocks of  $R$  in reverse order on even-numbered iterations of the outer loop. However, we could also maintain only one buffer for  $R$  and increase the number of buffers used for  $S$ . Which strategy yields the fewest disk I/O's?

## 15.8 Algorithms Using More Than Two Passes

While two passes are enough for operations on all but the largest relations, we should observe that the principal techniques discussed in Sections 15.4 and 15.5 generalize to algorithms that, by using as many passes as necessary, can process relations of arbitrary size. In this section we shall consider the generalization of both sort- and hash-based approaches.

### 15.8.1 Multipass Sort-Based Algorithms

In Section 11.4.5 we alluded to how the two-phase multiway merge sort could be extended to a three-pass algorithm. In fact, there is a simple recursive approach to sorting that will allow us to sort a relation, however large, completely, or if we prefer, to create  $n$  sorted sublists for any particular  $n$ .

Suppose we have  $M$  main-memory buffers available to sort a relation  $R$ , which we shall assume is stored clustered. Then do the following:

**BASIS:** If  $R$  fits in  $M$  blocks (i.e.,  $B(R) \leq M$ ), then read  $R$  into main memory, sort it using your favorite main-memory sorting algorithm, and write the sorted relation to disk.

**INDUCTION:** If  $R$  does not fit into main memory, partition the blocks holding  $R$  into  $M$  groups, which we shall call  $R_1, R_2, \dots, R_M$ . Recursively sort  $R_i$  for each  $i = 1, 2, \dots, M$ . Then, merge the  $M$  sorted sublists, as in Section 11.4.4.

If we are not merely sorting  $R$ , but performing a unary operation such as  $\gamma$  or  $\delta$  on  $R$ , then we modify the above so that at the final merge we perform the operation on the tuples at the front of the sorted sublists. That is,

- For a  $\delta$ , output one copy of each distinct tuple, and skip over copies of the tuple.
- For a  $\gamma$ , sort on the grouping attributes only, and combine the tuples with a given value of these grouping attributes in the appropriate manner, as discussed in Section 15.4.2.

When we want to perform a binary operation, such as intersection or join, we use essentially the same idea, except that the two relations are first divided into a total of  $M$  sublists. Then, each sublist is sorted by the recursive algorithm above. Finally, we read each of the  $M$  sublists, each into one buffer, and we

perform the operation in the manner described by the appropriate subsection of Section 15.4.

We can divide the  $M$  buffers between relations  $R$  and  $S$  as we wish. However, to minimize the total number of passes, we would normally divide the buffers in proportion to the number of blocks taken by the relations. That is,  $R$  gets  $M \times B(R)/(B(R) + B(S))$  of the buffers, and  $S$  gets the rest.

### 15.8.2 Performance of Multipass, Sort-Based Algorithms

Now, let us explore the relationship between the number of disk I/O's required, the size of the relation(s) operated upon, and the size of main memory. Let  $s(M, k)$  be the maximum size of a relation that we can sort using  $M$  buffers and  $k$  passes. Then we can compute  $s(M, k)$  as follows:

**BASIS:** If  $k = 1$ , i.e., one pass is allowed, then we must have  $B(R) \leq M$ . Put another way,  $s(M, 1) = M$ .

**INDUCTION:** Suppose  $k > 1$ . Then we partition  $R$  into  $M$  pieces, each of which must be sortable in  $k - 1$  passes. If  $B(R) = s(M, k)$ , then  $s(M, k)/M$ , which is the size of each of the  $M$  pieces of  $R$ , cannot exceed  $s(M, k - 1)$ . That is:  $s(M, k) = Ms(M, k - 1)$ .

If we expand the above recursion, we find

$$s(M, k) = Ms(M, k - 1) = M^2 s(M, k - 2) = \dots = M^{k-1} s(M, 1)$$

Since  $s(M, 1) = M$ , we conclude that  $s(M, k) = M^k$ . That is, using  $k$  passes, we can sort a relation  $R$  if  $B(R) \leq s(M, k)$ , which says that  $B(R) \leq M^k$ . Put another way, if we want to sort  $R$  in  $k$  passes, then the minimum number of buffers we can use is  $M = (B(R))^{1/k}$ .

Each pass of a sorting algorithm reads all the data from disk and writes it out again. Thus, a  $k$ -pass sorting algorithm requires  $2kB(R)$  disk I/O's.

Now, let us consider the cost of a multipass join  $R(X, Y) \bowtie S(Y, Z)$ , as representative of a binary operation on relations. Let  $j(M, k)$  be the largest number of blocks such that in  $k$  passes, using  $M$  buffers, we can join relations of  $j(M, k)$  or fewer total blocks. That is, the join can be accomplished provided  $B(R) + B(S) \leq j(M, k)$ .

On the final pass, we merge  $M$  sorted sublists from the two relations. Each of the sublists is sorted using  $k - 1$  passes, so they can be no longer than  $s(M, k - 1) = M^{k-1}$  each, or a total of  $Ms(M, k - 1) = M^k$ . That is,  $B(R) + B(S)$  can be no larger than  $M^k$ , or put another way,  $j(M, k) = M^k$ . Reversing the role of the parameters, we can also state that to compute the join in  $k$  passes requires  $(B(R) + B(S))^{1/k}$  buffers.

To calculate the number of disk I/O's needed in the multipass algorithms, we should remember that, unlike for sorting, we do not count the cost of writing the final result to disk for joins or other relational operations. Thus, we use  $2(k-1)(B(R) + B(S))$  disk I/O's to sort the sublists, and another  $B(R) + B(S)$

disk I/O's to read the sorted sublists in the final pass. The result is a total of  $(2k - 1)(B(R) + B(S))$  disk I/O's.

### 15.8.3 Multipass Hash-Based Algorithms

There is a corresponding recursive approach to using hashing for operations on large relations. We hash the relation or relations into  $M - 1$  buckets, where  $M$  is the number of available memory buffers. We then apply the operation to each bucket individually, in the case of a unary operation. If the operation is binary, such as a join, we apply the operation to each pair of corresponding buckets, as if they were the entire relations. For the common relational operations we have considered — duplicate-elimination, grouping, union, intersection, difference, natural join, and equijoin — the result of the operation on the entire relation(s) will be the union of the results on the bucket(s). We can describe this approach recursively as:

**BASIS:** For a unary operation, if the relation fits in  $M$  buffers, read it into memory and perform the operation. For a binary operation, if either relation fits in  $M - 1$  buffers, perform the operation by reading this relation into main memory and then read the second relation, one block at a time, into the  $M$ th buffer.

**INDUCTION:** If no relation fits in main memory, then hash each relation into  $M - 1$  buckets, as discussed in Section 15.5.1. Recursively perform the operation on each bucket or corresponding pair of buckets, and accumulate the output from each bucket or pair.

### 15.8.4 Performance of Multipass Hash-Based Algorithms

In what follows, we shall make the assumption that when we hash a relation, the tuples divide as evenly as possible among the buckets. In practice, this assumption will be met approximately if we choose a truly random hash function, but there will always be some unevenness in the distribution of tuples among buckets.

First, consider a unary operation, like  $\gamma$  or  $\delta$  on a relation  $R$  using  $M$  buffers. Let  $u(M, k)$  be the number of blocks in the largest relation that a  $k$ -pass hashing algorithm can handle. We can define  $u$  recursively by:

**BASIS:**  $u(M, 1) = M$ , since the relation  $R$  must fit in  $M$  buffers; i.e.,  $B(R) \leq M$ .

**INDUCTION:** We assume that the first step divides the relation  $R$  into  $M - 1$  buckets of equal size. Thus, we can compute  $u(M, k)$  as follows. The buckets for the next pass must be sufficiently small that they can be handled in  $k - 1$  passes; that is, the buckets are of size  $u(M, k - 1)$ . Since  $R$  is divided into  $M - 1$  buckets, we must have  $u(M, k) = (M - 1)u(M, k - 1)$ .

If we expand the recurrence above, we find that  $u(M, k) = M(M - 1)^{k-1}$ , or approximately, assuming  $M$  is large,  $u(M, k) = M^k$ . Equivalently, we can perform one of the unary relational operations on relation  $R$  in  $k$  passes with  $M$  buffers, provided  $M \leq (B(R))^{1/k}$ .

We may perform a similar analysis for binary operations. As in Section 15.8.2, let us consider the join. Let  $j(M, k)$  be an upper bound on the size of the smaller of the two relations  $R$  and  $S$  involved in  $R(X, Y) \bowtie S(Y, Z)$ . Here, as before,  $M$  is the number of available buffers and  $k$  is the number of passes we can use.

**BASIS:**  $j(M, 1) = M - 1$ ; that is, if we use the one-pass algorithm to join, then either  $R$  or  $S$  must fit in  $M - 1$  blocks, as we discussed in Section 15.2.3.

**INDUCTION:**  $j(M, k) = (M - 1)j(M, k - 1)$ ; that is, on the first of  $k$  passes, we can divide each relation into  $M - 1$  buckets, and we may expect each bucket to be  $1/(M - 1)$  of its entire relation, but we must then be able to join each pair of corresponding buckets in  $M - 1$  passes.

By expanding the recurrence for  $j(M, k)$ , we conclude that  $j(M, k) = (M - 1)^k$ . Again assuming  $M$  is large, we can say approximately  $j(M, k) = M^k$ . That is, we can join  $R(X, Y) \bowtie S(Y, Z)$  using  $k$  passes and  $M$  buffers provided  $M^k \geq \min(B(R), B(S))$ .

### 15.8.5 Exercises for Section 15.8

**Exercise 15.8.1:** Suppose  $B(R) = 20,000$ ,  $B(S) = 50,000$ , and  $M = 101$ . Describe the behavior of the following algorithms to compute  $R \bowtie S$ :

\* a) A three-pass, sort-based algorithm.

b) A three-pass, hash-based algorithm.

! **Exercise 15.8.2:** There are several “tricks” we have discussed for improving the performance of two-pass algorithms. For the following, tell whether the trick could be used in a multipass algorithm, and if so, how?

a) The hybrid-hash-join trick of Section 15.5.6.

b) Improving a sort-based algorithm by storing blocks consecutively on disk (Section 15.5.7).

c) Improving a hash-based algorithm by storing blocks consecutively on disk (Section 15.5.7).