Lock Granularity: SOfie database systeIlls allow programmers to override the default mechanisllt for choosing a lock granularity. For exalnple, 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 nUlnber 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 systenl 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 froIn the database and writing to a private workspace.

2. Validation: If the transaction decides that it wants to c0l111uit, the DBIvIS 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 deterrnines that there are no possible confliets, the changes to data objects 111ade by the transaction in its private workspace are copied into the database.

If, indeed, there are few confiicts, and validation can be done efficiently, this approach should lead to better performance than locking. If there are rnany conflicts, the cost of repeatedly restarting transactions (thereby wasting the work they've done) hurts perfornlance significantly.

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

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

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

Further, the first condition allows Tj to see some of Ti's changes, but clearly, they execute completely in serial order with respect to each other. The second condition allows Tj to read objects while Ti is still modifying objects, but there is no conflict because Tj does not read any object rnodified by Ti. Although Tj might overwrite some objects written by Ti, all of Ti's writes precede all of Tj's writes. The third condition allows Ti and Tj to write objects at the same time and thus have even IT10re 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 rnust also be completed (so that its effects are visible outside its private workspace) before other transactions can be validated.

A synchronization rnechanisrn such as a critical section can be used to ensure that at most one transaction is in its (colllbined) Validation/Write phases at any tirne. (When a process is executing a critical section in its code, the systern suspends all other processes.) Obviously, it is irnportant to keep these phases as short H.S possible in order to rniniruize the irnpact on concurrency. If copies of rnodified objects have to be copied frorn the private workspace, this can rnake the \Vrite phase long. An alternative approach (which carries the penalty of poor physical locality of objects, such as B·+· tree leaf pages, that rnust be clustered) is to use a level of indirection. In this schernc, every object is accessed via a logical pointer, and in the Write phase, we sirnply 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 optiInistic 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 frorn the private workspace. Sirnilarly, the irnplicit cost of blocking in a lock-based approach is replaced by the implicit cost of the work wasted by restarted transactions.

Improved Conflict Resolution[1]

Optirnistic 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, *Ti* cannot write any object read by *Tj*. IIowever, since the validation is airned at ensuring that *Ti* logically executes before *Tj*, there is no harm if *Ti* writes all data items required by *Tj* before *Tj* reads theIn.

The problerIl arises because we have no way to tell when *Ti* wrote the object (relative to *Tj*'s reading it) at the tirne we validate *Tj*, since all we have is the list of objects written by *Ti* and the list read by *Tj*. Such false conflicts can be alleviated by a finer-grain resolution of data conflicts, using rnechanisrI1s very sinlilar to locking.

The basic idea is that each transaction in the Read pha.se tells the DBMS about iteIIls it is reading, and ·when a transaction *Ti* is cornrnitted (and its writes are accepted), the DBMS checks whether any of the iterns written by *Ti* are being read by any (yet to be validated) transaction *Tj*. If so, we know that *Tj*'s validation rnust eventually fail. We can either allow *T,i* to discover this when it is validated (the die policy) or kill it and restart it innnediately (the kill policy).

The details are as follo\vs. Before reading a data iterrl, (1, transaction Tenters an access entry in a hash table. The access entry contains the *transaction id*, a *data object id*, and a *rn..odified* flag (initially set to false), and entries are hashed on the data object id. A terl1porary exclusive lock is obtained on the

[1] We thank Alexander Thomasian for writing this section.

hash bucket containing the entry, and the lock is held \vhile the read data iteIll is copied frolll the database buffer into the private 'workspace of the transactioll.

During validation of T the hash buckets of all data objects accessed by T are again locked (in exclusive 11lode) to check if T has encountered any data conflicts. T has encountered a conflict if the *rnodified* 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 lnodified by T, retrieve all access entries for this object, set the *rnodified* 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, 1nore irnportant, that access invariance prevails; that is, successive executions of T require the same data for execution. When T is restarted its execution tirne is nluch 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 optirrlistic concurrency control, a timestamp ordering is irnposed on transactions and validation checks that all conflicting actions occurred in the saIne order.

Tinlcstarnps can also be used in another \vay: Each transaction can be assigned a tirnestanlp at startup, and we can ensure, at execution tirne, that if action ai of transaction Ti conflicts \vith action aj of transaction Tj, ai occurs before aj if $TS(Ti) < TS(Tj)$. If an action violates this ordering, the transaction is aborted and restarted.

To irnplernent this concurrency control scheme, every database object 0 is given a read tirnestamp$RTS(0)$ and a write timestamp $WTS(0)$. If transaction T wants to read object 0, and $TS(T) < WTS(O)$, the order of this read with respect to the most recent write on 0 would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted *with a new, larger timestarnp.* If $TS(T) > WTS(O)$, Treads 0, 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 saIne conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention, where transactions are restarted with the *same* timestarnp 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 0:

1. If $TS(T) < RTS(O)$, the write action conflicts with the most recent read action of 0, 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 0 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 0 and $WTS(O)$ is set to $TS(T)$.

The Thomas Write Rule

We now consider the justification for the TholIlas Write Rule. If $TS(T) < WTS(O)$, the current write action has, in effect, been made obsolete by the rnost recent write of 0, which *follows* the current write according to the tirnestalnp ordering. We can think of T's write action as if it had occurred irnrnediately *before* the rnost recent write of 0 and was never read by anyone.

If the Thomas vVrite Rule is not used, that is, T is aborted in case (2), the tirnestamp protocol, like 2PL, allows only conflict serializable schedules. If the Tho1l1aS Write R,ule is used, some schedules are perrnitted that are not conflict serializable, as illustrated by the schedule in Figure 17.6.[2] Because $T2$'s write follows T1's read and precedes $T1$'s write of the sanle object, this schedule is not conflict serializable.

21n the other direction, 2PL pennits some schedules that are not allowed by the timestamp algorithm with the Thomas Write Rule; see Exercise 17.7.

T1	*T2*
$R(A)$	
	$W(A)$
	Cornrnit
$W(A)$	
COlnmit	

Figure 17.6 A Serializable Schedule 'rhat 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 $T8(T2) = 2$, this schedule is permitted by the timestalnp protocol (with or without the 1"ho111as Write Rule). The tiInestalnp protocol can be modified to disallow such schedules by buffering all write actions until the transaction COIDlnits. In the example, when TI wants to write A, $WTS(A)$ is updated to reflect this action, but the change to A is not carried out irrllnediately; instead, it is recorded in a private workspace, or buffer. When $T2$ wants to read A subsequently, its thnestamp is cornpared with $I¥TS(A)$, and the read is seen to be perrnissible. However, $T2$ is blocked until $T1$ cornpletes. If $T1$ cornrnits, its change to A is copied fro111 the buffer; other\vise, the changes in the buffer are discarded. $T2$ is then allowed to read A.

This blocking of $T2$ is sinlilar to the effect of $T1$ obtaining an exclusive lock on A. Nonetheles8, even with this modification, the tirnestarnp protocol perrnits sorne schedules not perrnitted by 2PL; the two protocols are not quite the same. (See Exercise 17.7.)

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

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 tilnestamps, thnestamp 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 Ti read the most recent version whose timestarnp precedes $TS(Ti)$.

If transaction 1'i wants to write an object, we must ensure that the object has not already been read by sonle other transaction Tj such that $TS(Ti) <$ 1'S(Tj). If we allow Ti to write such an object, its change should be seen by Tj for serializability, but obviously Tj, which read the object at SaIne tinle in the past, will not see Ti's change.

To check this condition, every object also has an associated read timestarnp, and whenever a transaction reads the object, the read timestamp is set to the maxhuuru of the current read tilnestarnp and the reader's tirnestarnp. If 7',t wants to write an object 0 and $TS(Ti) < RTS(O)$, Ti is aborted and restarted with a new, larger timestamp. Otherwise, Ti creates a new version of 0 and sets the read and write tirnestarnps of the new version to 7'S(Ti).

The drawbacks of this sehenle are similar to those of tirnestarnp concurrency control, and in addition, there is the cost of rnaintaining versions. On the other hand, reads are never blocked, which can be irnportant for workloads dorninated by transactions that only read values frorn the database.

What Do Real Systems Do? IBM DB2, Informix, Microsoft SQL Server, and Sybase ABE use Strict 2PL or variants (if a transaction requests a lower than SERIALIZABLE SQL isolation level; see Section 16.6). IVlicrosoft SQL Server also supports rnodifieation timestamps so that a transaction can run without setting locks and validate itself (do-it-yourself OptirnisticConC1:1rrency Control!). Oracle 8 uses a lllultiversion concurrency control scherne 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 systerlls support rnultiple-granu1arity 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?* Ilow 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 cornpare deadlock detection a,nd deadlock prevention schernes. Why are detection schernes rnore cornrnonly used? (Section 17.4)
- If the collection of database objects is not fixed, but can gro\v and shrink through insertion and deletion of objects, we lnust deal with a subtle corllplication known as the *phantorn problern.* Describe this problern and the index locking approach to solving the probleln. (Section 17.5.1)
- In tree index structures, locking higher levels of the tree can becorne a perforrnanee bottleneck. Explain why. I)escribe specialized locking techniques that address the problenl, and explain why they work correctly despite not lJeing two-phase. (Section 17.5.2)
- *Multiple-granularity locki'ng* enables us to set locks on objects that contain other objects, thus implicitly locking all contained objects. Why is this approach irnportant and how does it work? (Section 17.5.3)

- In *optimistic concurrency control*, no locks are set and transactions read and rnodify data objects in a private workspace. How are conflicts between transactions detected and resolved in this approach? (Section 17.6.1)
- In *tirnestamp-based concurrency control*, transactions are assigned a timestarnp at startup; how is it used to ensure serializability? How does the *Thomas Write Rule* improve concurrency? (Section 17.6.2)
- Explain why tinlestamp-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 tinwstanlps 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 serialiable.
8. Motivate and describe the use of locks for improved conflict resolution in Optinlistic Concurrency Control.

Exercise 17.2 Consider the following classes of schedules: *serializable, confiict-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; assurne that abort or cornrnit lllust follow all the listed actions.

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

3. Tl:R(X), T2:R(Y), T3:W(X), T2:R(X), Tl:R(Y)
4. Tl:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), Tl:W(X), T2:R(Y)
5. Tl:R(X), T2:W(X), Tl:W(X), T2:Abort, Tl:Cmnmit
6. Tl:R(X), T2:W(X), T1:W(X), T2:Comrnit, Tl:Comm.it
7. T1:W(X), T2:R(X), 1'l:W(X), 1'2:Abort, T.1:COllllllit
8. Tl:W(X), T2:R(X), Tl:W(X), T2:Conunit, Tl:Col1unit
9. Tl:W(X), T2:R(X), Tl:W(X), T2:Commit, Tl:Abort
10. 1'2: R(X), 1'3:W(X), T3:Cmnrnit, Tl:W(Y), Tl:Commit, T2:R(Y), T2:W(Z), T2:Colllmit
11. Tl:R(X), T2:W(X), T2:Cornrnit, Tl:W(X), Tl:Colllmit, T:3:R(X), T3:Collnnit
12. Tl:R(X), T2:W(X), Tl:W(X), T3:R(X), Tl:Comlllit, T2:Corn111it, 1'3:Comlnit

Exercise 17.3 Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Tilnestamp without the Thomas Write Rule, 1'ilnestamp 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, assurne that the timestamp for transaction *Ti* 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 81: Tl:R(X), T2:W(X), T2:W(Y), T3:W(Y), Tl:W(Y), Tl:Commit, T2:Commit, T3:Commit
- Sequence 82: Tl:R(X), T2:W(Y), T2:W(X), T3:W(Y), Tl:W(Y), Tl:C0111mit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control rnechanislns, describe how the concurrency control mechanislll handles the sequence.

Assurne that the tirnestarnp of transaction *Ti* is *i*. Fbr lock-based concurrency control rnechaniS111S, 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 reslllned; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

1. Strict 2PL with tiluestamps 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. Tiruestarup concurrency control with buffering of reads and writes (to ensure recoverability) and the Tholnas Write Rule.
6. rvluitiversioll concurrency control.

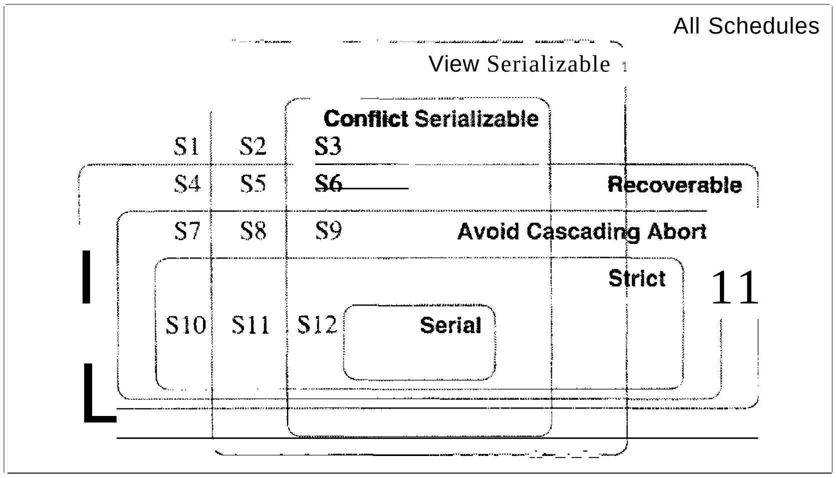


Figure 17.9 Venn Diagram for Classes of Schedules

Exercise 17.5 For each of the following locking protocols, assulning that every transaction follows that locking protocol, state which of these desirable properties are ensured: serializability, conflict-·serializability, recoverability, avoidance of cascading aborts.

1. Always obtain an exclusive lock before writing; hold exclusive locks until end-of-transaction. No shared locks are ever obtained.
2. In addition to (1), obtain a shared lock before reading; shared locks can be released at any time.
3. As in (2), and in addition, locking is two-phase.
4. As in (2), and in addition, all locks held until end-of-transaction.

Exercise 17.6 The Venn diagranl (frorn [76]) in Figure 17.9 shows the inclusions between several classes of schedules. Give one exaulple schedule for each of the regions S1 through *S12* in the diagrarn.

Exercise 17.7 Briefly answer the following questions:

1. Draw a Venn diagram that ShO\V8 the inclusions between the classes of schedules perulittccl by 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.*
2. Give one example schedule for each region in the diagrarIl.
3. Extend the Venn diagranl to include serializable and conflict-serializable schedules.

Exercise 17.8 Answer each of the follmving questions briefly. The questions are based on the following relational schern.a:

Ernp(*eid:* <u>integer</u>, *ename:* string, *age:* integer, *salary:* real, *did:* integer)
Dept(<u>*did:* integer</u>, *dname:* string, *flooT:* integer)

and on the fc)llowing update cornrnand:

replace (salary = 1.1 * EMP.salary) where EMP.ename = 'Santa'

1. Give an example of a query that would conflict with this comrnand (in a concurrency control sense) if both were run at the same tim.e. Explain what could go wrong, and how locking tuples would solve the probleIll.
2. Give an exarnple of a query or a cOHlInand that would conflict with this cOIl1ruand, such that the conflict could not be resolved by just locking individual tuples or pages but requires index locking.
3. Explain what index locking is and how it resolves the preceding conflict.

Exercise 17.9 SQL supports four isolation-levels and two access-rllodes, for a total of eight cornbinations of isolation-level and access-rnode. Each corubinatioll inlplicitly defines a class of transactions; the follO\ving questions refer to these eight classes:

1. For each of the eight classes, describe a locking protocol that allows only transactions in this class. Does the locking protocol for a given class make any assurnptiolls about the locking protocols used for other classes? Explain briefly.
2. Consider a schedule generated by the execution of several SQL transactions. Is it guaranteed to be conflict-serializable? to be serializable'? to be recoverable?
3. Consider a schedule generated by the execution of several SQL transactions, each of which has READ ONLY access-mode. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
4. Consider a schedule generated by the execution of several SQL transactions, each of which has SERIALIZABLE isolation-level. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
5. Can you think of a tinlCstarup-based concurrency control scheme that can support the eight classes of SQL transactions?

Exercise 17.10 Consider the tree shown In Figure 19.5. Describe the steps involved in executing each of the following operations according to the tree-index concurrency control algorithm discussed in Section 19.3.2, in terms of the order in which nodes are locked, unlocked, read, and written. Be specific about the kind of lock obtained and answer each part independently of the others, always starting with the tree shown in Figure 19.5.

1. Search for data entry 40*.
2. Search for all data entries $k*$ with $k \leq 40$.
3. Insert data entry 62*.
4. Insert data entry 40*.
5. Insert data entries 62* and 75*.

Exercise 17.11 Consider a database organized in tenns of the following hierarachy of objects: The database itself is an object *(D)*, and it contains two files *(FI* and *F'2)*, each of which contains 1Q00 pages *(PI* ... PlOOO ancl *P*1001 ... P2000, respectively). Each page contains 100 records, and records are identified as $p : i$, where p is the page identifier and i is the slot of the record on that page.

I'vlultiple-granularity locking is used, with S, X, IS,$^{\prime}I$X and $S'IX$ locks, and database-level, file-level, page-level and record-level locking. For each of the following operations, indicate the sequence of lock requests that 1nust be generated by a transaction that wants to carry out (just) these operations:

1. Read record P1200 : 5.
2. Read records $P1200$: 98 through P1205 : 2.
3. Read all (records on all) pages in file $F1$.
4. 'Read pages P500 through P520.
5. Read pages PIO through P980.
6. Read all pages in *PI* and (based on the values read) rnodify 10 pages.
7. Delete record P1200 : 98. (This is a blind write.)
8. Delete the first record frorn each page. (Again, these are blind writes.)
9. Delete all records.

Exercise 17.12 Suppose that we have only two types of transactions, Tl and *T2*. Transactions preserve database consistency when run individually. We have defined several *integrity constraints* such that the DBMS never executes any SQL statenwnt that brings the database into an inconsistent state. Assunle that the DBlVIS does not perform *any* concurrency control. Give an exarllple schedule of two transactions Tl and T2 that satisfies all these conditions, yet produces a database instance that is not the result of any serial execution of $T1$ and T2.

BIBLIOGRAPHIC **NOTES**

Concurrent access to B trees is considered in several papers, including [70,456,472,505,678]. Concurrency control techniques for Linear Hashing are presented in [240] and [543]. Multiple-granularity locking is introduced in [3:36] and studied further in [127, 449].

A concurrency control method that works with the ARIES recovery rnethod is presented in [545]. Another paper that considers concurrency control issues in the context of recovery is [492]. AlgorithrIls for building indexes without stopping the DBMS are presented in [548] and [9]. The performance of B tree concurrency control algorithnls is studied in [704]. Perfornlance of various concurrency control algorithms is discussed in [16, 729, 735]. A good survey of concurrency control rnethods and their perfornlance is [734]. [455] is a comprehensive collection of papers on this topic.

TilYwstarIlp-based multiversion concurrency control is studied in [6201. IvIultiversion concur--rency control algoritllIIls are studied forrnally in [87J. Lock-based rnultiversion techniques are considered in [460]. Optirnistic concurrency control is introduced in [457]. The use of access invariance to irnprove conflict resolution in high-contention environrnents is discussed in [281] and [280]. Transaction management issues for real-tirne database systerns are discussed in [1, 15, 368, :382, 386, 448]. There is a large body of theoretical results on database concurrency control; [582, 89] offer thorough textbook presentations of this rnaterial.

18

CRASH RECOVERY

- What steps are taken in the ARIES method to recover fronl a DBMS crash?
- How is the log rnaintained during nonnal operation?
- How is the log used to recover frorn a crash?
- What infonnation in addition to the log is used during recovery?
- What is a checkpoint and why is it used?
- W'hat happens if repeated crashes occur during recovery?
- How is media failure handled?
- How does the recovery algorithnl 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, cornrnit, abort, end, cOlnpensation; transaction table, lastLSN; dirty page table, recLSN; checkpoint, fuzzy checkpointing, master log record; rnedia recovery; interaction with concurrency control; shadow paging

Hurnpty Durnpty sat on a \vall.
lIurnpty Durnpty had a great fall.
A.ll the King's horses and all the King's tnen
Could not put lIllrnpty together again.

—Old nursery rhyrne

The recovery manager of a DBMS is responsible for ensuring two irnportant properties of transactions: Atornicity and durability. It ensures *atomicity* by undoing the actions of transactions that do not conlIllit and *durability* by rnaking sure that all actions of conunitted transactions survive system crashes (e.g., a core durnp caused by a bus error) and Inedia failures (e.g., a disk is corrupted).

1'"he recovery rnanager is one of the hardest cOlllponents of a DBMS to design and inlplernent. It rnust deal 'with a wide va,riety of database states because it is called on during systenl failures. In this chapter, we present the ARIES recovery algorithnl, which is conceptually sinlple, works well with a wide range of concurrency control rnechanisrns, and is being used in an increasing number of database syterns.

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 frorn 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 algorithrn designed to work with a steal, no-force approach. When the recovery rnanager 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 tiITle of the crash.
2. Redo: Repeats all actions, starting frOID an appropriate point in the log, and restores the database state to what it was at the tirne of the e1'a8h.
3. lJndo: Undoes the actions of transactions that did not cOllunit, so that the database reflects only the actions of cornrnitted transactions.

Consider the sirnple execution history illustrated in Figure 18.1. When the systeIIl is restarted, the A,nalysis phase identifies *'Tl* and *T3* as transactions

LSN	LOG
10	update: T1 **writes** P5
20	update: T2 writes P3
30	T2 **commit**
40	T2end
50	update: T3 writes PI
60	update: T3 writes P3
	CRASH, RESTART

Figure 18.1 Execution History with a Crash

active at the time of the crash and therefore to be undone; *T2* as a corrnuitted transaction, and all its actions therefore to be written to disk; and PI, *P3,* and *P5* as potentially dirty pages. All the updates (including those of TI and *T3)* are reapplied in the order shown during the Redo phase. Finally, the actions of TI and *T*3 are undone in reverse order during the Undo phase; that is, *T3's* write of *P3* is undone, *T*3's write of PI is undone, and then TI's write of *P5* is undone.

Three Inain principles lie behind the ARIES recovery algoritlun:

- Write-Ahead Logging: Any change to a database object is first recorded in the log; the record in the log lllUst 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 systern 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 tirne of the crash (effectively aborting theln).
- Logging Changes During Undo: Changes lnada 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 frorn other recovery algorithrns and is the basis for rnuch of its sirnplicity and flexibility. In particular, ABIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The seCOJlc! and third points are also

Crash Recovery: IBM DB2, Inforrnix, Microsoft SQL Server, Oracle 8, and Sybase l\SE all use a WAL seherue for recovery. IBIvI DB2 uses ARIES, and the others use seherues that are actually quite sinlilar 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 THELOG

The log, SOlnetirnes 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 sinlultaneously lost is negligibly small.

The most recent portion of the log, called the log tail, is kept in nlain Inemory 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 ruonotonically increasing order; this property is required for the ARIES recovery algorithrn. If the log is a sequential file, in principle growing indefinitely, the LSN can sirllply be the address of the first byte of the log record.[1]

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

A log record is\vritten for each of the following actions:

[1]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 I.ISN may be the log record id plus a *wrap-count*.

- Updating a Page: After rTlodifying the page, an *update* type record (described later in this section) is appended to the log tail. Tlhe pageLSN of the page is then set to the LSN of the update log record. (The page Blust be pinned in the buffer pool while these actions are carried out.)

- Conl1nit: When a transaction decides to conunit, 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 cOllunit record.[2] The transaction is considered to have cOIlnnitted at the instant that its cOlnmit log record is written to stable storage. (Solne additional steps rnust be taken, e.g., reilloving the transaction's entry in the transaction table; these follow the writing of the cOlInnit 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 comrnitted, some additional actions rnust be taken beyond writing the abort or COllllllit log record. After all these additional steps are c()lnpleted, 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 frorn a crash), its updates are undone. When the action described by an update log record is undone, a *cornpensation 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 rnaintained as a linked list going back in tirne, using the prevLSN field; this list HUlst be updated whenever a log record is added. The transII) 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 rnentioned 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. frhe pageID field is the page iel of the Inodified page; the length in bytes and the offset of the

[2]Note that this step requires the buffer manager to be able to selectively *force* pages to stable storage.

prevLSN	transID	type	pageID	length	offset	before-image	after-image

Fields common to all log records   Additional fi.elds for update log records

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-iluage; similarly an undo-only update record contains just the before-iluage.

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 nor-rnal system execution when a transaction is aborted or during recovery froIn a crash.) A cOlnpensation 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. 'fhe cornpensation 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 11pdate record *U*; this field in C is set to the value of prevLSN in *[J.*

As an exarllple, consider the fourth update log reeord shown in Figure 18.3. If this update is undone, a CLR would be written, and the infornation in it would include the transII), pageID, length, offset, and before-image fields froln the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-irnage value; thus, this value and the location of the affected bytes constitute the redo infonnation for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 18.:3.

lJnlike an update log record, a CLR describes an action that \vill never be *undone*, that is, we never undo an undo action. '1'he reason is sirnple: An update log record describes a change lnade by a transaction during nonnal execution and the transaction rnay subsequently be aborted, whereas a CLR describes an a,ctiol1 taken to rollback a transaction for which the decision to abort has already been rnade. 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 a1nount of space needed for the log during restart froin a crash: The nUlnber of CLRs that ca,n be written during LJndo is no 1nore than the nurnber of update log records for active transactions at the tirne of the crash.

A CLR II1ay be written to stable stora,ge (follo\ving WAL, of course) but the undo action it describes rIlay not yet been written to disk when the systenl 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 infonnation needed to reapply, 01' redo, the change described but not to reverse it.

18.3 OTHER RECOVERY-RELATED STRU'CTURES

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

- Transaction Table: This table contains one entry for each active transaction. 'The entry contains (arnong other things) the transaction id, the status, and a field called **lastLSN,** which is the LSN of the rnost recent log record for this transaction. The status of a transaction can be that it is in progress, corunlitted, or aborted. (In the latter two cases, the transaction will be rernoved fro1l1 the table once certain 'clean up' steps are c(nupleted.)

- 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 becorne dirty. Note that this LSN identifies the earliest log record that lnight have to be redone for this page during restart fronl a crash.

I)uring norrnal operation, these are rnainta..ined by the transaction rnanager and the buffer rnanager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the follc)\ving silupic exarnple. Transaction TIOOO changes the value of bytes 21 to 23 011 page P500 frorn 'ABC' to 'DEF', transaction 'T2000 changes 'HIJ' to 'KLM' on page P600, transaction $T2000$ changes bytes 20 through 22 fronl 'GDE' to 'QRS' on page $P500$, then transaction T1000 changes 'TUV' to 'WXY' on pageP505. The dirty page table, the transaction table,[3] and

[3]The status field is not shown in the figure for space reasons; all transactions are in progress.

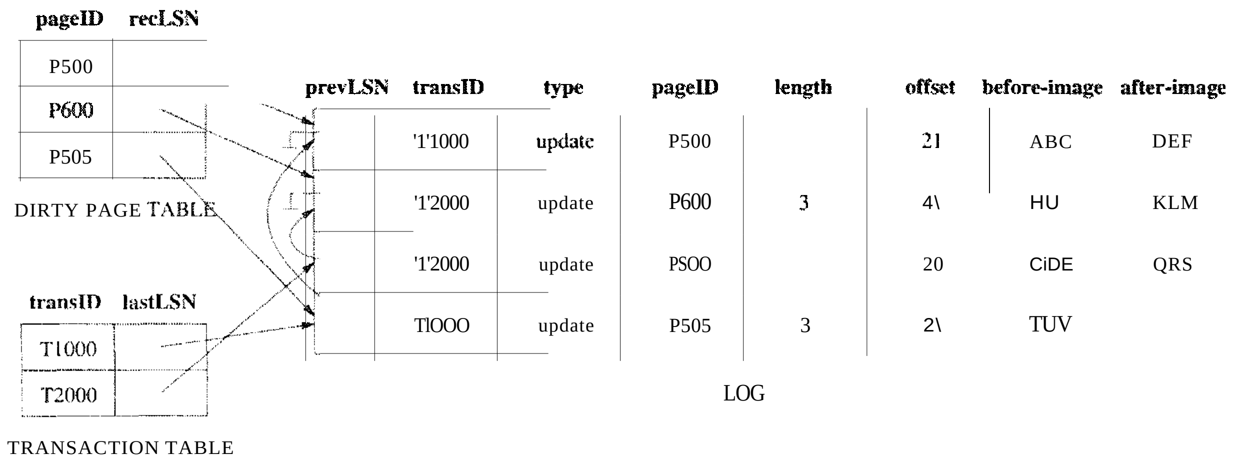


Figure 18.3 Instance of Log and Ttansaction Table

the log at this instant are shown in Figure 18.3. ()bserve that the log is shown growing froni top to bottorn; 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 iInportant---for exarnple, T2000's change to page P500 follows TIOOO's change to page *P500,* and in the event of a crash, these changes nUlst be redone in the sanle 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 rnust be forced to stable storage. This is accornplished 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 irnportance of the WAL protocol carulot be overerl1phasized- --\VAL is the fundarnentaJ rule that ensures that a record of every change to the database is available while atternpting to recover froni a crash. If a transaction rnade (l. change and committed, the no-force approach Incans that some of these changes rnay not have been written to disk at the tirne of a sulJsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a cornl11.itted transaction survive crashes. Note that the definition of a *committed transaction* is effectively 'a transa,ction all of whose log records, including a conunit record, have 1)een written to stable storage'.

When a transaction is cornrnitted, 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 a,ctions taken under a force approach: If a force approach is used, all the pages rIlodified by the transaction, rather than a portion of the log that includes all its records, IHIIS!, be forced to disk when the transaction conllIlits. The set of all changed pages is typically 11luch 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 Inuch s1na11er than the page size. Further, the log is 1naintained as a sequential file, and all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is luuch sIllaller than the cost of \vriting all changed pages to disk.

18.5 CHECKPOINTIN(;

A checkpoint is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBl\1S can reduce the alnount 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 *begirLcheckpoint* 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 SystCIll or writing out pages in the buffer pool (unlike some other forlns of checkpointing). On the other hand, the effectiveness of this checkpointing technique is lirnited by the earliest recLSN of pages in the d.irty pages table, because during restart we Inust redo changes starting froin the log record \vhose LSN is equal to this recI.lSN. l-Iaving a background process that periodically writes dirty pages to disk helps to lirnit this probleln.

When the SystCIIl cornes back up after a crash, the restart process begins by locating the rnost recent checkpoint record. For uniforlnity, the systeIll always begins no1'n1al execution by takirlg a checkpoint, in which the transaction table and dirty page table are both Clnpty.

18.6 RECOVERING FROM A SYSTEM CRASH

\Vhen the systenl is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 18.4.

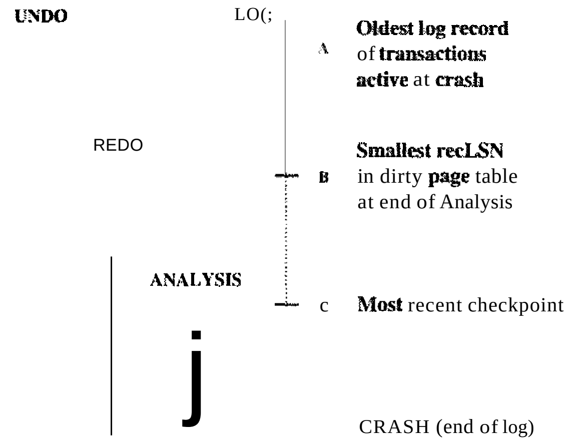


Figure 18.4 Three Phases of Restart in ARIES

The Analysis phase begins by examInIng the rnost recent begin_checkpoint record, whose LSN is denoted C in Figure 18.4, and proceeds forward in the log until the last log record. '1'he Redo phase follows Analysis and redoes all changes to any page that Illight have been dirty at the tir11e of the crash; this set of pages and the starting point for Redo (the srnallest recLSN of any dirty page) are deterrnined during Analysis. 'The Undo phase follows Redo and undoes the changes of all transactions active at the tirne 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 lllost recent change first.

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

18.6.1 Analysis Phase

The Analysis phase perfonns three tasks:

1. It detennines the point in the log at which to start the Redo pass.
2. It deterrnines (a conservative superset of the) pages in the buffer pool that were clirty at the tirne of the crash.
3. It identifies transactions that were active at the tirne of the crash and rnust be undone.

Analysis begins by exEtrnining the rnost recent begirLcheckpoint 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 tilne of the checkpoint.

(If additional log records are between the begiILcheckpoint and encLcheckpoint records, the tables HIUst be adjusted to reflect the inforluation in these records, but \ve cnnit the details of this step. See Exercise 18.9.) A.naJysis then scans the log in the for\vard direction until it reaches the end of the log:

- If an end log record for a transaction T is encountered,T is reIlloved fronl 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 rnodified:
  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a cOllnnit 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 tilue of the crash—this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the tirne of the crash but rnay also contain SOlne pages that were written to disk. If an *end_write* log record were written at the cornpletion of ea,ch write operation, the dirty page table constructed during Analysis could be lnade rnore accurate, but in AHJES, the additional cost of writing eneLwrite log records is not considered to be worth the gain.

As an exarnple, consider the execution illustrated in Figure 18.3. Let us extend this execution by assurning that $T2000$ COlIllnits, then *TIOnO* rnodifies another page, say, .P700, and appends an update record to the log tail, and then the systern crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in rnernory, are lost in the crash. The rnost recent checkpoint was taken at the beginning of the execution, \vith an ernpty tran.saction table and dirty page table; it is not shown in Figure 18.3. After examining this log record, \vhich we assurne is just before the first log record shown in the figure, Analysis initializes the two tables to l>e ernpty. Scanning forward in the log, T'1000 is added to the transaction table; in additiol1,P500 is added to the dirty page ta,blc\vith recLSN equal to the LSN of the first sho\vn log record. Sirnilarly, T2C)OO is added to the transaction table andPGOO 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 eOllnnit record forT2000 (not in the figure) is no\v encountered, and T2000 is relIloved fro111 the transaction table.

The Analysis phase is now eornplete, and it is recognized that the only active transaction at the time of the crash is *TIOOO*, \vith lastLSN equal to the LSN of the fourth record in Figure 18.3. rrhe 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······--thecorresponding change to page *P700* cannot have been written to disk either!

SaIne of the updates rnay have been written to disk; for concreteness, let us assurne that the change to *P600* (and only this update) was written to disk before the crash. ThereforeP600 is not dirty, yet it is included in the dirty page table. rIhe 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, coill-rnitted 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 vVAL-based recovery algoritlnIls and causes the database to be brought to the sarne state it was in at the time of the crash.

rrhe R,edo phase begins with the log record that has the srnallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that rnay not have been written to disk prior to the crash. Starting frorn this log record, R,edo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action HUlst be redone. The action rnust be redone unless one of the follo\ving conditions holds:

- The affected page is not in the dirty page table.
- rrhe 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.
- 1'he pageLSN (stored on the page, which rnust be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously 1118a11S that all changes to this page have been written to disk. Because the recLSN is the first update to this page that lnay not have been written to disk, the second condition rneans 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 assurnption is irnportant here!)

If the logged action lllust 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 exarnple discussed in Section 18.6.1. FrorIl 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 assurned 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 exarnines the second log record. Again, the affected page, P600, is fetched and the pageLSN is cornpared to the LSN of the update log record. In this case, because we assurned thatP600 was written to disk before the crash, they are equal, and the update does not have to be redone.

The rernaining log records are processed sirnilarly, bringing the systern back to the exact state it was in at the tirue of the crash. Note that the first two conditions indicating that a redo is unnecessary never hold in this exaruple. Intuitively, they corne into play when the dirty page table contains a very old recLSN, going back to before the rJlost recent checkpoint. In this case, as Redo scans forwa.rd frorn 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. Sorne of these pages Inay 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 t\VO conditions allow us to recognize this situation \vithout fetching the page. (The reader is encouraged to construct exaulples 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 rCllloved '£1'01n the transaction table.

18.6.3 Undo Phase

The Undo phase, unlike the other two phases, scans backward fronl the end of the log. The goal of this phase is to undo the actions of all transactions active at the tilne of the crash, that is, to effectively abort the1n. This set of transactions is identified in the transaction table constructed by the AIlalysis 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 1110st recent log record (the lastLSN field) for each such transaction. Such transactions are called loser transactions. All actions of losers IllUst be undone, and further, these actions rnust 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 (Le., rnost recent) LSN value in this set and processes it, until rro1Jndo is ernpty. 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 cornpletely undone, and the CLR, is discarded.
2. If it is an. update record, a CLR, is written and the corresponding a,ction 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 rroUndo is empty, the lJndo phase is cornplete. I{estart is no\v cornplete, and the systenl can proceed with nonnal operations.

Let us continue with the scenario discussed in Sections 18.6.1 and 18.6.2. The only active transaction at the tiTne of the crash was detennined to be TI0.00. :F'rom the transaction table, we get the LSN of its Inost recent log record, which is the fourth update log record in Figure 18.3. '1'he update is undone, and a CLR is \vritten\vith 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 a,nel an end log record for $T1000$ are written, and the IJndo phase is cornplete.

In this exarnple, undoing the action recorded in the first log record causes the action of the third log record, \vhich is due to a conunitted traJlsaetioIl, to be overwritten and thereby lost! rrhis situation arises because $T2000$ overwrote a data itcrIl \vritten by TIOOO while $T1000$ was still active; if Strict 2PLwere followed, $T2000$ would not have been allowed to overwrite this data iterH.

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 exarnple in Figure 18.5, discussed next, illustrates this point.

Crashes during Restart

It is important to understand how the lTndo algorithrn presented in Section 18.6.3 handles repeated systern 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 systern crashes using an execution history, shown in Figure 18.5, that abstracts away unnecessary detail. This exarnple 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.

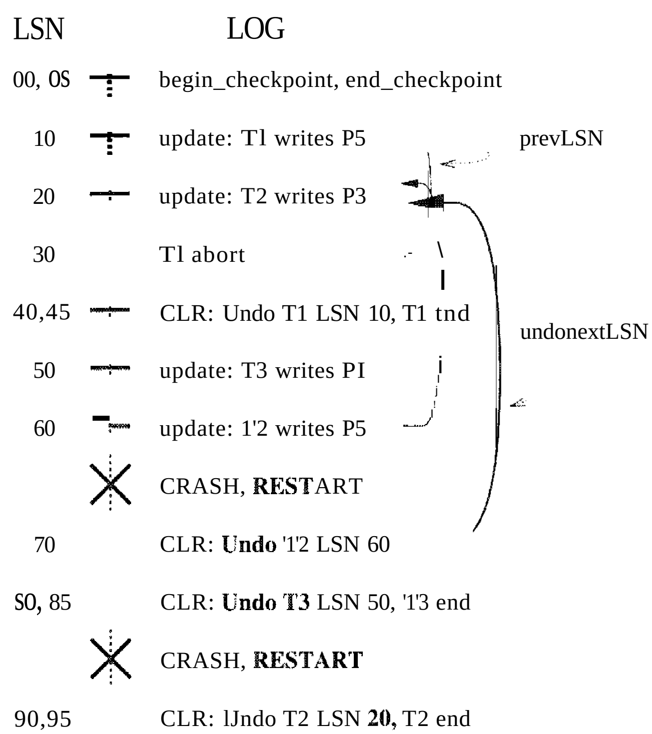


Figure 18.5 Example of Undo with Repeated Crashes

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, SOIne special value used in the prevLSN field of the first log record for a, transaction to indicate tha,t there is no previous log record. We also cOlnpacted the figure by occasionally displaying two log records (separated by a cOIlllna) on a single line.

Log record (with LSN) 30 indicates that *Tl* aborts. All actions of this transaction should be undone in reverse order, and the only action of $T1$, described by the update log record 10, is indeed undone as indicated by CLR, 40.

After the first crash, Analysis identifies F)l (with recLSN 50), $P3$ (with recLSN 20), and *P5* (with recLSN 10) as dirty pages. Log record 45 shows that *Tl* is a cornpleted transaction; hence, the transaction table identifies *T2* (with lastLSN 60) andT3 (with lastLSN 50) as active at the tirne of the crash. '1'he Redo phase begins with log record 10, which is the rninirnurn recLSN in the dirty page table, and reapplies all actions (for the update and CLR, records), as per the Redo algorithIl1 presented in Section 18.6.2.

The r1'olJndo set consists of LSNs 60, for $T2$, and 50, for $T3$. The lJndo 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 llndoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for 1 12. Now the largest rernaining LSN in the ITOUndo set is 50. The write corresponding to log record 50 is now undone, and a CLH, describing the change is 'written. rrhis CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction *T3*. Therefore $T3$ is cOITlpletely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the systern crashes a second tirHe; however, the changes described by these records ITlay not have been written, to disk..

When the systern is resta.rted after the s8c011(1 crash. Analysis deterrnines that the only active transactioIl at the time of the crash was *'T2;* in addition, the dirty pa,ge table is identical to what it was during the previous restart. Log records 10 througll 85 are processed again during Redo. (If sorne of the changes made during the previous Redo were written to disk, the pageLSN's on the affected pages are used to detect this situation and avoid writing these pages again.) The lJndo phase considers the onlyLSN in the TolJndo set, 70, and processes it }))" adding tIle llndoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed l)y undoing $T2$'s write of page $P3$, a.nd a CLR is written (LSN 90). Because 20 is the first of 7'2'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 *T2,* al1d the TolJndo set is now empty.

Recovery is no\v complete, and norrnal execution can resurne with the ·writing of a checkpoint record.

This exarnple illustrated repeated crashes during the lJndo phase. For corn-pleteness, let us consider what happens if the system crashes while R,estart 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 sa1lle infornration as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that sorne of the changes rnade during Redo may have been written to disk prior to the crash. R,estart starts again with the Analysis phase and then the Redo phase, and sorne update log records that were redone the first tirne around will not be redone a second tirne 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 rninirnize 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 rnaking a copy of the database. Be-cause copying a large database object such as a file can take a long tirHe, and the I)BMS rnust be allowed to continue with its operations in the Ineantirne, creating a copy is handled in a rnanner sirnilar 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 cornnlitted transactions and undo the changes of uncoI1unitted transactions (as of the tirne of the rnedia recovery operation).

The begin_checkpoint LSN of the rnost recent cOlllplete checkpoint is recorded along with the copy of the database object to luinirnize the work in reapplying changes of committed transactions. Let us COlnpare the smallest recLSN of a dirty page in the corresponding encLcheckpoint record \vith the I;SN of the begirLcheckpoint record and call the slua.ller of these two LSNs *I*. We observe that the actions recorded in all log records with LSNs less than *I* Inust be reflected in the copy. Thus, 0111y log records with LSNs greater than *I* need be reapplied to the copy.

Finally, the updates of transactions that are incornplete at the tiIl1e of Inedia recovery or that were aborted after the fuzzy copy was corllpleted need to be undone to ensure that the page reflects only the actions of conunitted transactions. The set of such transactions can be identified as in the Analysis pass, and we ornit the details.

18.8 OTHER APPROACHES AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the Inost popular alternative recovery algoritllln s also rnaintain a log of database actions according to the WAL protocol. A InajaI' 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 rnodifications. For exalllple, 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 Inay also insert and delete entries frorn the tree. If record-level locks are set rather than page-level locks, the entry 15* I11ay be on a different physical page when T aborts fr0111 the one that T inserted it into. In this case, the undo operation for the insert of 15* lllUSt be recorded in logical tenns 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 rnore locks 1nust be set). Hence, there is a trade-off between different WAL-based recovery schclnes. We chose to cover ARIES because it has several attractive properties, in particular, its sirnplicity and its ability to support fine-granularity locks and logging of logical operations.

One of the earliest recovery algorithrns, llsed in the Syster11 R prototype at IBl\'1, 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 thTough a **page table**, which maps page ids to disk addresses. When a transaction Inakes changes to a data page, it actually Inakes a copy of the page, called the **shadow** of the page, a,nel changes the shadow page. The transaction copies the appropriate part of the page table and chan,ges the entry for the changed page to point to the shadow, so that it can see the changes; ho\vever, other transactions continue to see the original page table, and therefore the original page, until this transaction COll1lnits. Aborting a transaction is sirnple: .Just discard its shadow versions of the page table and the data pages. Cornrnitting a transaction involves rnaking its version of the page table public and discarding the original data pages that are superseded by shado\v pages.

This schelue suffers frorn a nUlnber of problerlls. First, data becornes highly fragrnented clue to the replacernent of pages by shadow versions, which rIlay be located far fr01n the original page. This phenornenon reduces data clustering and rnakes good garbage collection irnperative. Second, the sche1ne does not yield a sufficiently high degree of concurrency. rrhird, 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 rllust be specially handled because the sernantics of aborting an abort transaction gets rnurky.

For these reasons, even in Systern R, shadow paging was eventually superseded by \VAL-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 algoritluu? (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)
- \Vhat are the different types of log records and when are they written? (Section 18.2)
- What inforrnation is rnaintained in the transaction table and the dirty page table? (Section 18.3)
- What is Write-Ahead Logging? \Vhat is forced to disk at the tirne a transaction COIlllnits? (Section 18.4)
- What is a fuzzy checkpoint? \Vhy is it useful? What is a master log record? (Section 18.5)
- In \vhich direction does the .A.nalysis phase of recovery scan the log? At \vhich point in the log does it begin and end the scan? (Section 18.6.1)
- Descril)c \vhat infonnation is gathered in the Analysis phase and ho\v. (Section 18.6.1)

- In \vhich 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? \Vhat 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 111edia 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 rnanager ensure atornicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a systenl crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

Exercise 18.2 Briefly answer the follO\ving 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. VVhat are redoal)le 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_cbeckpoint
20	update: Tl writes P5
30	update: T2 writes P3
40	T2 commit
SO	T2end
60	update: T3 writes P3
70	Tl abort
	CRASH, RESTART

Figure 18.6 Execution with a Crash

LSN	LOG
00	update: Tl writes P2
10	update: Tl writes PI
20	update: T2 writes P5
30	update: T3 writes P3
40	T3 commit
50	update: T2 writes PS
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 Figllre 18.7.

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

LSN	LOG
00	**begin_checkpoint**
10	end_checkpoint
20	update: 1'1 **writes** PI
30	update: 1'2 writesP2
40	update: 1'3 writes P3
50	1'2 commit
60	update: 1'3 writes P2
70	1'2 end
80	update: 1'1 writes P5
90	1'3 abort
	CRASH,RESTART

Figure 18.8 Execution with Multiple Crashes

3. Show the log after *T2* 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 systerll 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. \Vhat is done during Undo?
,5. Show the log when recovery is complete, including all non-null prevLSN and unclonextLSN 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 systerll so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and trallsaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointiug a,pproach of ARIES?
3. What happens if a second begiILcheckpoint record is encountered during the Analysis phase?
4. C;an a second en(Lcheckpoint record be encountered during the AnaJysis phase?
5. Why is the use of CLRs irnportant for the use of undo actions that are not the physical inverse of the original update?

LSN	LOG
00	**begin_checkpoint**
10	**update: T1** writes PI
20	'1'1 commit
30	update: **T2** writes P2
40	'1'1 end
50	'1'2 abort
60	update: '1'3 **writes** P3
70	end_checkpoint
80	'1'3 commit
	CRASH, RESTART

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 rrlaximum nunlber 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 cOInpletes 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. \Vhy 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 rnade the sirnplifying assulTlptioll that no log records appeared between the begill-checkpoint and end_checkpoint records for the Inost recent cOlnplete checkpoint. The following questions explore how such records should be handled.

1. Explain why log records could be written between the begiIl-checkpoint and eneLcheckpoint records.
2. Describe how the Analysis phase could be Inodified to handle such records.
3. Consider the execution sho\vn in Figure 18.9. Show the contents of the encLcheckpoint record.
4. Illustrate your rnodified Analysis phase on the execution shown in Figure 18.9.

Exercise 18.10 Answer the following questions briefly:

1. Explain how 1nedin recovery is handled in ARIES.
2. What are the pros and cons of using fuzzy durnps for media recovery?
3. What are the sirYlilarities and differences between checkpoints and fuzzy chunps?
4. Contrast ARIES with other WAL-based recovery schernes.
5. Contrast AHlES with shadow-page-based recovery.

BIBLIOGRAPHIC NOTES

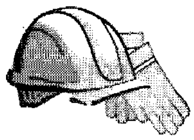
Our discussion of the ARIES recovery algorithm is based on [544]. [282] is a survey article that contains a very readable, short description of ARIES. [541, 545] also discuss ARIES. Fine,·granularity locking increases concurrency but at the cost of 11101'e locking activity; [542] suggests a technique based on LSNs for alleviating this problerYl. [458] presents a for111al verification of ARIES.

[355] is an excellent survey that provides a broader treatrnent of recovery algoritlulls than our coverage, in which we chose to concentrate on one particular algorithrn. [17] considers perfor-rnance of concurrency control and recovery algorithrIls, taking into account their interactions. The irnpact of recovery on concurrency control is also discussed in [769]. [625] contains a performance analysis of various recovery techniques. [236] cornpares recovery techniques for main rnerllory database systeulS, which are optirnized for the case that 11l0st of the active data set fits in rnain H1ernory.

[478] presents a description of a recovery algorithm based on write-ahead logging in which 'loser' transactions are first undone and then (only) transactions that corl1nlitted before the crash are redone. Shadow paging is described in [493, 337]. A scherne that uses a cOlnbination of shadow paging and in-place updating is described in [624].

PART VI

DATABASE DESIGN AND TUNING



19

SCHEMA REFINEMENT AND NORMAL FORMS

- What problems are caused by redundantly storing information?
- What are functional dependencies?
- What are nornlal forms and what is their purpose?
- What are the benefits of BCNF and 3NF?
- What are the considerations in decolllposing relations into appropriate normal forms?
- Where does normalization fit in the process of database design?
- Are luore general dependencies useful in database design?
- Key concepts: redundancy, insert, delete, and update anomalies; functional dependency, Armstrong's Axioms; dependency closure, attribute closure; normal fonns, BCNF, 3NF; decOlnpositions, lossless-join, dependency-preservation; multivalued dependencies, join dependencies, inclusion dependencies, 4NF, 5NF

> It is a nlelancholy truth that even great InelI have their poor relations.
>
> Charles Dickens

Conceptual database design gives us a set of relation schemas and integrity constraints (ICs) that can be regarded as a good starting point for the final database design. T'his initial design IHust be refined by taking the lCg into account rnore fully than is possiblc\vith just the ER rnodel constructs aIId also by considering perforrnance criteria and typical workloads. In this chapter, we c!iscllss how lCs can be used to refine the conceptual schema produced by translating an ER 1Hodel design into a collection of relations. \Vorkload and perforrnance considerations are discussed in Chapter 20.

We concentrate on an irnportant class of constraints called *functional dependencies*. Other kinds of les, for exarnple, *multivalued dependencies* and *join dependencies*, also provide useful inforrnation. They can sOIuetilnes reveal redundancies that cannot be detected using functional dependencies alone. We discuss these other constraints briefly.

This chapter is organized as follows. Section 19.1 is an overview of the schenla refineInent approach discussed in this chapter. We introduce functional dependencies in Section 19.2. In Section 19.3, we show how to reason with functional dependency information to infer additional dependencies from a given set of dependencies. We introduce norlnal forIns for relations in Section 19.4; the normal form satisfied by a relation is a measure of the redundancy in the relation. A relation with redundancy can be refined by *decomposing it,* or replacing it with smaller relations that contain the saIne information but without redundancy. We discuss deco1npositions and desirable properties of decompositions in Section 19.5, and we show how relations can be decomposed into smaller relations in desirable normal forms in Section 19.6.

In Section 19.7, we present several examples that illustrate how relational schemas obtained by translating an ER model design can nonetheless suffer froln redundancy, and we discuss how to refine such schemas to eliminate the problems. In Section 19.8, we describe other kinds of dependencies for database design. We conclude with a discussion of nornlalization for our case study, the Internet shop, in Section 19.9.

19.1 INTRODUCTION TO SCHEMA REFINEMENT

We now present an overview of the probleIns that schenla refinement is intended to address and a refinernent approach based on decolnpositions. Iledundant storage of inforrnation is the root cause of these problerns. Although decoInposition can elirninate redundancy, it can lead to problclns of its own and should be used with caution.

19.1.1 Problems Caused by Redundancy

Storing the same inforrnation redundantly, that is, in l110re than one place \vithin a database, can lead to several problcll1S:

- Redundant Storage: SOU1C iuforInation is stored repeatedly.

- **Update Anomalies:** If one copy of sueh repeated data is updated, an inconsistency is created unless all copies are sirnilarly updated.
- **Insertion Anomalies:** It may not be possible to store certain inforlnation unless sorne other, unrelated, inforIIlatioIl is stored as well.
- **Deletion Anomalies:** It rnay not be possible to delete certain inforrnation without losing sOHle other, unrelated, infofrnation as well.

Consider a relation obtained by translating a variant of the IIourly_Emps entity set frorn Chapter 2:

Hourly_Emps(<u>*ssn*</u>, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

In this chapter, we ornit attribute type inforrnation for brevity, since our focus is on the grouping of attributes into relations. We often abbreviate an attribute narne to a single letter and refer to a relation schema by a string of letters, one per attribute. For exarllple, we refer to the Hourly_Ernps scherna as *SNLRWH* (*W* denotes the *hourly_wages* attribute).

1'he key for Hourly_Emps is *ssn*. In addition, suppose that the *hourly_wages* attribute is deterrnined by the *rating* attribute. That is, for a given *rating* value, there is only one perrllissible *hourly_wages* value. This IC is an exanlple of a *functional dependency*. It leads to possible redundancy in the relation Hourly_Ernps, as illustrated in Figure 19.1.

	narne	*lot*	*rating*	*hourly_wages*	*hours_worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Sruiley	22	8	10	30
131-24-3650	Srllethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Figure 19.1 An Instance of the Hourly_Emps Relation

If the same value appears in the *rating* colurnn of two tuples, the IC tells us that the sarne value HUlst appear in the *hourly_wages* colurnn as well. This redundancy has the sarne negative consequences as before:

- *Redundant Storage:* rrhe rating value 8 corresponds to the hourly wage 10, and this association is repeated three tirnes.
- *[Tpdate Anomalies:* The *hourly_wages* in the first tuple could be updated without rnaking a sirnilar change in the second tuple.

- *Insertion Anomalies:* We cannot insert a tuple for an crnployee unless \ve know the hourly wage for the ernployee's rating value.
- *Delet'ion Anomalies:* If we delete all tuples with a given rating value (e.g., we delete the tuples for Snlcthurst and Guldu) we lose the association between that *rating* value and its *hourly_wage* value.

Ideally, we want schemas that do not pennit redundancy, but at the very least we want to be able to identify schernas that do allow redundancy. Even if we choose to accept a scherna with sorne of these drawbacks, perhaps owing to perforlnance considerations, we want to rnake an infonned decision.

Null Values

It is worth considering whether the use of *null* values can address some of these problems. As we will see in the context of our exarnple, they cannot provide a complete solution, but they can provide sorne help. In this chapter, we do not discuss the use of *null* values beyond this one exarnple.

Consider the example Hourly_Elnps relation. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. For instance, to deal with the insertion anolnaly exarnple, we can insert an elTIplayee tuple with *null* values in the hourly wage field. However, *null* values cannot address all insertion anornalies. For exarnple, we cannot record the hourly wage for a rating unless there is an ernployee with that rating, because we cannot store a null value in the *ssn* field, which is a prirnary key field. Sinlilarly, to deal with the deletion anomaly exarnple, we rnight consider storing a tuple with *null* values in all fields except *rating* and *hourly_wages* if the last tuple with a given *rating* would otherwise be deleted. However, this solution does not work because it requires the *8871,* value to be *null,* and prirnary key fields cannot be *null.* Thus, *null* values do not provide a general solution to the problerns of reclundancy, even though they can help in sorne cases.

19.1.2 Decompositions

Intuitively, redundancy arises when a relational schcrna forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other Ies) can 'be used to identify such situations and suggest re£1nernents to the schema. The essential idea is that rnany problerns arising fr0111 redundancy can be addressed by replacing a relation 'with a collection of 'smaller' relations.

A. decomposition of a relation schema R consists of replacing the relation scherna by tvvo (or 1no1'e) relation schcrnas that each contain a subset of the attributes of R and together include all attributes in R. Intuitively, we want to store the inforrnation in any given instance of R by storing projections of the instance. This section exalnines the use of decornpositions through several exanlples.

We can decornpose lIourly_Ernps into two relations:

> Hourly_Emps2(<u>*ssn*</u>, *naTne, lot, rating, hours_worked*)
> \Vages(<u>*rating*</u>, *hourly_wages*)

The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 19.1 is shown in Figure 19.2.

ssn	*narne*	*lot*	*rating*	*hours_worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Sluiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	*hourly_wages*
8	10
5	7

Figure 19.2 Instances of Hourly_Emps2 and vVages

Note that we can easily record the hourly wage for any rating sirnply by adding a tuple to Wages, even if no ernployee with that rating appears in the current instance of flourly_Ernps. Changing the wage associated \vith a rating involves updating a single Wages tuple. This is rnore efficient than updating several tuples (as in the original design), and it elirninates the potential for inconsistency.

19.1.3 Problems Related to Decomposition

lJnless we are careful, decornposing a relation scherna can create 1n01'e problerns than it solves. Two irnportant questions llHlst be asked repeatedly:

1. 1)0 we need to decornpose a relation?

2. What problerns (if any) does a given deeornposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation scherna is ill one of these nOfrual 1'orrns, we know that certain kinds of problerlls cannot arise. Considering the norrnal forrn of a given relation scherna can help us to decide \vhether or not to decornpose it further. If we decide that a relation scherna 111USt be decornposed further, we rnust choose a particular dec()lnposition (I.e., a particular collection of slnaller relations to replace the given relation).

With respect to the second question, two properties of decornpositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decornposed relation froln corresponding instances of the s111aller relations. The *dependency-preservation* property enables us to enforce any constraint on the original relation by sinlply enforcing SaIne contraints on each of the srnaller relations. That is, we need not perform joins of the slllaller relations to check whether a constraint on the original relation is violated.

From a performance standpoint, queries over the original relation may require us to join the decomposed relations. If such queries are common, the perfor-rnance penalty of decomposing the relation may not be acceptable. In this case, we may choose to live with some of the problems of redundancy and not decompose the relation. It is important to be aware of the potential problerns caused by such residual redundancy in the design and to take steps to avoid thern (e.g., by adding SaIne checks to application code). In sonle situations, decomposition could actually *improve* performance. This happens, for exam-ple, if lnost queries and updates exanline only one of the decornposed relations, which is srnaller than the original relation. We do not discuss the irnpact of decompositions on query perforInance in this chapter; this issue is covered in Section 20.8.

()ur goal in this chapter is to explain S0111e powerful concepts and design guide-lines based on the theory of functional dependencies. A good database designer should have a firm grasp of nor1nal fonns and \vhat problerns they (do or do not) alleviate, the technique of decornposition, and potential problerns with decornpositions. For example, a designer often asks questions such as these: Is a relation in a given nonnal forIn? Is a decornposition clependency-preserving? Our objective is to explain when to raise these questions and the significance of the answers.

19.2 FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of Ie that generalizes the concept of a *key*. Let *R* be a relation scherna and let X and Y be nonernpty sets of attributes in *R*. We say that an instance *r* of *R* satisfies the FD$X \rightarrow Y$ [1] if the following holds for every pair of tuples tl and *t2* in *r*:

If *t1.X* = *t2..X*, then *tl.}T* = *t2.Y*.

We use the notation tl.X to refer to the projection of tuple t1 onto the attributes in *X*, in a natural extension of our TRC notation (see Chapter 4) *t.a* for referring to attribute *a* of tuple t. An FD $X \rightarrow$ Yessentially says that if two tuples agree on the values in attributes *X*, they 111Ust also agree on the values in attributes *Y*.

Figure 19.3 illustrates the rneaning of the FD $AB \rightarrow$ C by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, *AB* is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the *A* field or the *B* field, they can differ in the C field without violating the FD. On the other hand, if we add a tuple (aI, bl, *c2, dl)* to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

A	*B*	*C*	*D*
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	dl
a2	bl	c3	ell

Figure 19.3 An Instance that Satisfies $AB \rightarrow C$

Recall that a *legal* instance of a relation nUlst satisfy all specified les, including all specified FDs. As noted in Section 3.2, Ies rIlust be identified and specified based on the sernantics of the real-world enterprise being n1odeled. By looking at an instance of a relation, we rnight be able to tell that a certain FD does *not* hold. I-Iowever; we can never deduce that an FD *docs* hold by looking at one or 1I10re instances of the relation, beca,use an FD, like other les, is a staternent about *all* possible legal instances of the relation.

[1] $X \rightarrow Y$ is read as X *functionally determines Y*, or simply as X *determines Y*.

.A prirnary key constraint is a special case of an FD. The attributes in the key play the role of X, and the set of all attributes in the relation plays the role of Y. Note, however, that the definition of an FD does not require that the set X be 111iniInal; the additionalrninimality condition Illust be Inet for X to be a key. If $X \rightarrow Y$ holds, \vhere Y is the set of all attributes, and there is SCHne (strictly contajned) subset V of X such that $V \rightarrow Y$ holds, then X is a *superkey*.

In the rest of this chapter, ·we see several exarIlples of FDs that are not key constraints.

19.3 REASONING ABOUT FDS

Given a set of FDs over a relation scheula .*R*, typically several additional FDs hold over *R* whenever all of the given FDs hold. As an exalnple, consider:

Workers(*ssn*, *naTne*, *lot*, *did*, *since*)

We know that *ssn* → *did* holds, since *ssn* is the key, and FD *did* → *lot* is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same *ssn* value, they Blust have the sarne *did* value (frolH the first FD), and because they have the sarrle *did* value, they must also have the saIne *lot* value (1'1'0111 the second FD). Therefore, the FD *ssn* → *lot* also holds on Workers.

We say that an FD f is implied by a given set F of FDs if f holds on every relation instance that satisfies all dependencies in F; that is, f holds whenever all FDs in F hold. Note that it is not sufficient for f to hold on SaIne instance that satisfies all dependencies in F; rather, f rnust hold on *every* instance that satisfies all dependencies in F.

19.3.1 Closure of a Set of FDs

The set of all .FDs irnplied by a given set F of FDs is called the closllre of **F**, denoted as F^+. An irnportant question is how we can infer, or cornpute, the closure of a given set F of FDs. The answer is sirnple and elegant. The following three rules, called Armstrong's Axioms, can be applied repeatedly to infer all FI)s irnplied by a set F of FDs. We use X, Y, and Z to denote *sets* of attributes over a relation scherna R:

- Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$.
- Augn1.entation: If)($\rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
- Transitivity: If)($\rightarrow Y$ and Y Z, then $X \rightarrow Z$.

Theorem 1 *Armstrong's Axioms are* **sound**, *in that they generate only FDs in F^+ when applied to a set* F *of FDs. They are also* **complete**, *in that repeated a]Jplicat'ion afthese rules will generate all FDs in the closure* .Fl+.

The soundness of Arrnstrong's Axiorns is straightfor\vard to prove. Cornpleteness is harder to show; see Exercise 19.17.

It is convenient to use SOlne additional rules while reasoning about $P+$:

- Union: If $X \rightarrow$ Yand $X \rightarrow Z$, then $X \rightarrow YZ$.
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow y'$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong's AxiolllS.

To illustrate the use of these inference rules for FDs, consider a relation schelua ABC with FDs $A \rightarrow B$ and $B \rightarrow$ C. In a trivial FD, the right side contains only attributes that also appear on the left side; such dependencies always hold due to reflexivity. Using reflexivity, we can generate all trivial dependencies, which are of the form:

$X \rightarrow$ Y, where $Y \subseteq X$, $X \subseteq ABC$, and $Y \subseteq ABC$.

FrOHl transitivity we get $A \rightarrow$ C. Fronl auglnentation we get the nontrivial dependencies:

$AC \rightarrow BC$, $AB \rightarrow AC$, $AB \rightarrow$ C13.

As another exalnple, we use a rnore elaborate version of Contracts:

Contracts(*contractid*, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *val'ue*)

\Ve denote the schenla for Contracts as $CSJDPQV$. The rneaning of a tuple is that the contract with *contractid* C is an agreement that supplier S (*supplierid*) 'will supply Q iterns of part? *(par-tid)* to project J (*projectid*) associated with departrnent D (*deptid*); the value V of this contract is equal to *value*.

The following res are known to hold:

1. The contract id C is a key: C $\rightarrow$ $CSJDP(JV$.
2. A project purchases a given part using a single contract: $.II) \rightarrow C$.

3. **A** departInent purchases at most one part froul a supplier: $8D \rightarrow P$.

Several additional FDs hold in the closure of the set of given FDs:

From $.IP \rightarrow C$, $C \rightarrow CSJDPQV$, and transitivity, we infer $.IP \rightarrow CSJDPQV$.

FraIn $8D \rightarrow P$ and augnlentation, we infer $SDJ \rightarrow JP$.

FraIn $8DJ \rightarrow .IP$, $JP \rightarrow CSJDPQV$, and transitivity, we infer $SDJ \rightarrow CSJD$-PQV. (Incidentally, while it Illay appear tenlpting to do so, we *cannot* conclude $SD \rightarrow CSDPQV$, canceling $.I$ on both sides. FD inference is not like aritlunetic Illultiplication!)

We can infer several additionalFDs that are in the closure by using augruentation or decomposition. For exarnple, from $C \rightarrow CSJDPQV$, using decomposition, we can infer:

$$C \rightarrow C,\ C \rightarrow 5,\ C \rightarrow J,\ C \rightarrow D, \text{ and so forth}$$

Finally, we have a number of trivial FDs from the reflexivity rule.

19.3.2 Attribute Closure

If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set F of FDs, we can do so efficiently without cornputing Fl+. We first cornpute the **attribute closure** X+with respect to F, \vhich is the set of attributes A such that $X \rightarrow A$ can be inferred using the Arrnstrong Axioms. The algorithrn for computing the attribute closure of a set X of attributes is shown in Figure 19.4.

```
closure = X;
repeat until there is no change: {
        if there is an FD U → V in F such that U ⊆ closure,
            then set closure == closure U V·
        }
```

Figure 19.4 Computing the Attribute Closure of Attribute Sct X

Theorem 2 *The algorithln shown in Figure 1.9.4 cornputes the attribute closure X-+- of the attribute set X 'IDith respect to the sct of FDs Fl.*

The proof of this theorern is considered in Exercise 19.15. This algoriUuIl can be rl10dified to find keys by starting with set X containing a, single attribute and stopping as soon as *closure* contains all attributes in the relation scherna. By varying the starting attribute and the order in which the algorithrIl considers FDs, we can obtain all candidate keys.

19.4 NORMAL **FORMS**

Given a relation sche111a, we need to decide whether it is a good design or we need to decornpose it into srnaller relations. Such a decision llUlst be guided by an understanding of what problenls, if any, arise froln the current schelna. To provide such guidance, several normal forms have been proposed. If a relation schelna is in one of these norrnal forIns, we know that certain kinds of problerlls cannot arise.

The nonnal forrns based on FDs are *first nor-rnal forrn* (1*NF)*, *second normal forrn (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF).* These fonns have increasingly restrictive requirernents: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in INF. A relation is in first normal fortH if every field contains only atornic values, that is, no lists or sets. This requirerllent is iInplicit in our definition of the relational rnode!. Although SOHle of the newer database systerlls are relaxing this requirernent, in this chapter we aSSUlne that it always holds. 2NF is Inainly of historical interest. 3NF and BCNF are irnportant frolH a database design standpoint.

While studying norrnal fonns, it is irnportant to appreciate the role played by FDs. Consider a relation scherna R with attributes ABC. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. ()n the other hand, suppose that we have the FI) $A \rightarrow$ 13. Now if several tuples have the sarne A value, they rnust also have tllC sarneB value. This potential redundancy can be predicted using the FD il1fonnation. If 11101'8 detailed 1Cs are specified, we rnay be able to detect rnore subtle redundancies as well.

We primarily discuss redundancy revealed l)y PI) inforrnation. In Section 19.8, we discuss 11lore sophisticated 1Cs called *multivalued dependencies* and *join dependencies* and norrnal forrns based on theIn.

19.4.1 Boyce...Codd Normal Form

Let R be a relation scherna, F be the set ofF'I)s given to hold over R, X be a subset of the attributes ofR, and A be an attribute of R. R is in Boyce-Codd normal form if, for everyFl)X → A in F, one of the follo\ving statements is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey.

Intuitively, in a BCNF relation, the only nontrivial dependencies are those in 'which a key detennines SaIne attribute(s). Therefore, each tuple can be thought of as an entity or relationship, identified by a key and described by the reluaining attributes. !(ent (in [425]) puts this colorfully, if a little loosely: "Each attribute nlust describe [an entity or relationship identified by] the key, the \vhole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure 19.5, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)

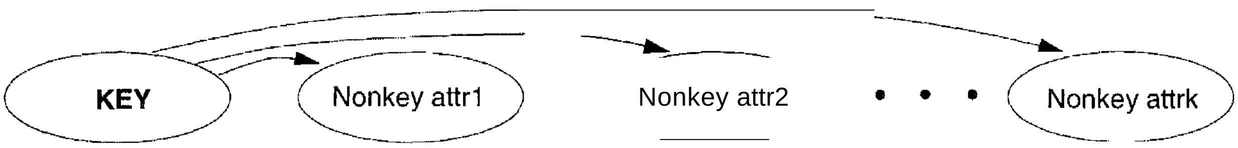


Figure 19.5 FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD infonnation alone. It is thus the Inost desirable norrnal form (fronl the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 19.6.

X	Y	A
x	y_1	a
x	y_2	?

Figure 19.6 Instance Illustrating BCNF

This figure shows (t\VO tuples in) an instance of a relation with three attributes X, Y, and A. There are two tuples with the saIne value in the X colurnn. Now suppose that we kno\v that this instance satisfies an FD $X \rightarrow A$. We can see that one of the tuples has the value a in the A colurnn. What can we infer al)out the value in the A colllrnn in the second tuple? 'Using the FI), \ve can conclude that the second tuple also has the value a in this colurnn. (Note that this is really the only kind of inference we can make about values in the fields of tuples by usingFDs.)

But is this situation not an exaInple of redundancy? \Ve appear to have stored the value a twice. Can such a situation arise in a BCNF relation? The ans\ver is No! If this relation is in BCNF, because A is distinct fronl X, it follows that X IllU8t be a key. (Otherwise, the FD $X \rightarrow A$ \vould violate BCNF.) If X is a key, then Yl = Y2, which Ineans that the two tuples are identical Since a relation is defined to be a *set* of tuples, we cannot have two copies of the saIne tuple and the situation shc)\vn in Figure 19.6 cannot arise.

rrherefore, if a relation is in BCNF, every field of every tuple records a piece of inforlnation that cannot be inferred (using only FDs) frorn the values in all other fields in (all tuples of) the relation instance.

19.4.2 Third Normal Form

Let R be a relation scherna, F be the set of FDs given to hold over R, X be a subset of the attributes of R, and A be an attribute of R. R is in **third normal forIn** if, for every FD $X \rightarrow A$ in F, one of the following statenlents is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey, or
- A is part of sorne key for R.

rrhe definition of 3NF is sinlilar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a rela,tion is a *minimal* set of attributes that uniquely deterrnines all other attributes. A rrlllst be part of a key (any key, if there are several). It is not enough for A to be part of a superkey, because the latter condition is satisfied by every attribute! Finding all keys of a relation scherna is known to be an NP-cornplete problern, and so is the prob1ern of detennining whether a relation seherna is in 3NF.

Suppose that a dependency $X \rightarrow A$ causes a violation of 3NF. There are two cases:

- X is *a proper subset of some key* K. Such a dependency is 801netirnes called **a partial dependency**. In this case, we store *(X, ./1)* pairs redundantl:y. As an example, consider the Reserves relation with attributes *SBDC* frorn Section 19.7.4. The only key is *8El)*, and we have the FD $S \rightarrow C$. We store the credit ca,rd nurnber for a sailor as lnany tirnes as there are reservations for that sailor.
- X is *not a proper subset of any key.* Such a dependerlcy is sornetirnes called a **transitive dependency**, because it rneans we have a chain of

dependencies $!(\to X \to A$. The problem is that we cannot associate an X value with a K value unless we also associate an A value with an X value. As an exanlple, consider the Hourly_Emps relation with attributes *SNLRWH* froIn Section 19.7.1. The only key is S, but there is an FD $R \to W$, which gives rise to the chain $S \to R \to W$. The consequence is that we cannot record the fact that elnployee S has rating R without knowing the hourly \vage for that rating. This condition leads to insertion, deletion, and update anoIllalies.

Partial dependencies are illustrated in Figure 19.7, and transitive dependencies are illustrated in Figure 19.8. Note that in Figure 19.8, the set X of attributes 11lay or Illay not have some attributes in conunon with KE-Y; the diagranl should be interpreted as indicating only that X is not a subset of KEY.

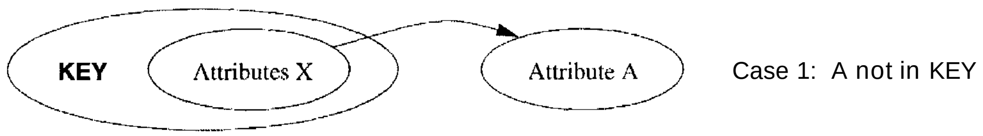


Figure 19.7 Partial Dependencies

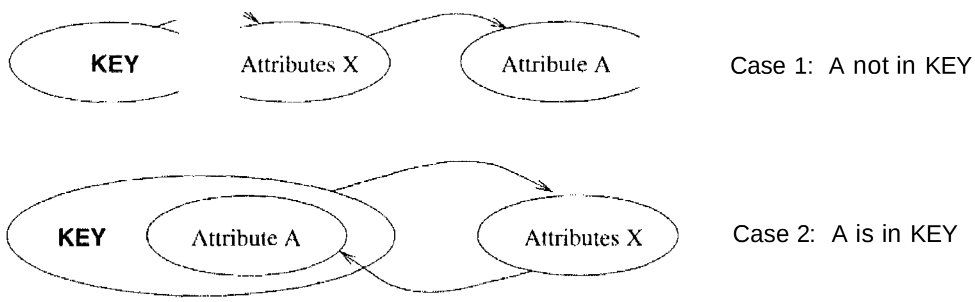


Figure 19.8 Transitive Dependencies

The Inotivation for 3NF is rather technical. By Inaking an exception for certain dependencies involving key attributes, we can ensure that every relation schclna can be decornposed into a collection of 3NF relations using only dec(nnpositions that have certain desirable properties (Section 19.5). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirernents just enough to Inake this guarantee possible. We Inay therefore cOlnprornise by settling for a 3NF design. As we see in Chapter 20, we 11lay sometimes accept this compromise (or even settle for a non-3NF scheIna) for other reasons as well.

lJnlike BCNF, however, BOlne redundancy is possible with 3NF. The problerns associated \vith partial and transitive dependencies persist if there is a nontrivial dependency $X \to A$ and X is not a superkey, even if the relation is in 3NF l)ccause A is pa,rt of a key. To understand this point, let us revisit the R,eserves relation with attributes *SEDe* and the FD $S \rightarrow C$, which states that a sailor uses a unique credit card to pay for reservations. S is not a key, and C is not part of a key. (In fact, the only key is *SED.)* Hence, this relation is not in 3NF; *(S, CJ* pairs are stored redundantly. IIowever, if we also know that credit cards uniquely identify the owner, we have the FD C $\rightarrow$ 5, which rneans that *CBD* is also a key for Reserves. Therefore, the dependency $S \rightarrow C$ does not violate 3NF, and R,eserves is in 3NF. Nonetheless, in all tuples containing the saIne 5 value, the saIne (8, *CJ* pair is redundantly recorded.

For cOlllpleteness, we reluark that the definition of second norrnal form is essentially that partial dependencies are not allowed. Thus, if a relation is in 3NF (which precludes both partial and transitive dependencies), it is also in 2NF.

19.5 PROPERTIES OF DECOMPOSITIONS

DecoIllposition is a tool that allows us to eliminate redundancy. As noted in Section 19.1.3, however, it is important to check that a decoInposition does not introduce new problellls. In particular, we should check whether a decomposition allows us to recover the original relation, and whether it allows us to check integrity constraints efficiently. We discuss these properties next.

19.5.1 Lossless-Join Decomposition

Let R be a relation schelna and let F be H, set of FDs over R. A decolnposition of R into two schernas with attribute sets X andY is said to be a lossless-join decomposition with respect to F if, for every instance r of R that satisfies the dependencies in F, $\pi_X(r) \bowtie \pi_Y(r) = \tau$. In other words, we can recover the original relation 1'rom the deconlposed relations.

This definition can easily be extended to cover a decornposition of R into more than two relations. It is easy to see that $\tau \subseteq \pi_X(r) \bowtie \pi_Y(r)$ ahvays holds. III general, though, the other direction does not hold. If we take projections of a relation and recornbine theln using natural join, we typically obta.in sOlne tuples that 'were not in the original relation. This situation is illustrated in Figure 19.9.

By replacing the instance r shown in Figure 19.9 with the instances $\pi_{SP}(r)$ and $\pi_{PI)}(r)$, we lose sorne information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that the relationships $(81, p_1, d_3)$ and (s_3, p_1, d_1) do not hold. rrhe decoluposition of schema *SPD* into *S.P* and *PI)* is therefore loss,Y if the instance r shown in the figure is legal, that is, if this

S	P	D
sl	pl.	ell
s2	p2	d2
s3	pl.	d3

Instance r

S	P
s1	pI
s2	p2
s3	pI

$\pi_{SP}(r)$

P	D
p1	d1
p2	d2
~~p1~~	~~d3~~

$\pi_{PD}(r)$

S	P	D
81	pI	ell
82	p2	d2
83	pI	d3
sl	pI	d3
s3	pI	ell

$\pi_{SP}(r) \bowtie \pi_{PD}(r)$

Figure 19.9 Instances Illustrating Lossy Decompositions

instance could arise in the enterprise being rIlodeled. (Observe the siInilarities between this example and the Contracts relationship set in Section 2.5.3.)

All decompositions used to eli'minate redundancy **must** *be lossless.* The following sirnple test is very useful:

Theorern 3 *Let* R *be a relation and* F *be a set of FDs that hold over* R. *The decomposition of* R *into relations with attribute sets* R_1 *and* R_2 *is l08sless if and only if* $p+$ *contains either the FD* R_1 n $R2 \rightarrow R_1$ *or the FD* R_1 n $R_2 \rightarrow R2$.

In other words, the attributes cornrIlon to *Rl* and *R2* HUlst contain a key for either RIOI' R_2.[2] If a relation is decornposed into 1110re than two relations, an efficient (time polynomial in the size of the dependency set) algoritllln is available to test whether or not the dec(nnposition is lossless, but we will not discuss it.

Consider the lIourly_Ernps relation again. It has attributes *SNLRWII*, and the FI) $R \rightarrow W$ causes a violation of 3NF. We dealt with this violation by decorIlposing the relation into *SNLRII* and *RW*. Since *R* is cornrnon to both decornposed relations and $R \rightarrow W$ holds, this decornposition is lossless-join.

This exarnple illustrates a general observation that follows froIH Theorerll 3:

> If an Ff) $X \rightarrow Y$ holds over a relation R and X n Y is ernpty, the decornposition of R into $R - Y$ and XY is lossless.

X appears in both $R - Y$ (since $X \cap Y$ is ernpty) and XY, and it is a key for XY.

[2]See Exercise 19.19 for a proof of Theorern 3. Exercise 19.11 illustrates that the 'only if' claim depends on the assumption that only functional dependencies can be specified as integrity constraints.

Another hnportant observation, which we state without proof, has to do with repeated decolnpositiollS. Suppose that a relation R is decornposed into Rl and $R2$ through a IOBsless-join decolupositiol1, and that $R1$ is decolnposed *intoRl.1* and $R12$ through another lossless-join decolnposition. Then, the decolnposition of R into $R11$, $R.12$, and $R2$ is lossless-join; by joining $R11$ and $R12$, \ve can recover $R.1$, and by then joining Rl and $R2$, we can recover R.

19.5.2 Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *C8JDPCJVfronl* Section 19.3.l. The given FDs are $C \rightarrow C8JDPQV$, $JP \rightarrow C$, and $SD \rightarrow P$. Because SD is not a key the dependency $SD \rightarrow P$ causes a violation of BCNF.

We can decolnpose Contracts into two relations with schelnas $CSJDQV$ and SDP to address this violation; the decolnposition is lossless-join. There is one subtle problelll, however. We can enforce the integrity constraint $JP \rightarrow C$ easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same JP values (as the inserted tuple) but different C values. Once we decompose Contracts into $CSJDQV$ and SDP, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into $CSJDQV$. We say that this decornposition is not dependency-preserving.

Intuitively, a *dependency-preserving decornposition* allows us to enforce all FDs by exarnining a single relation instance on each insertion or rnodification of a tuple. (Note that deletions cannot cause violation of FDs.) To define dependency-preserving decornpositions precisely, we have to introduce the concept of a projection of FDs.

Let R be a relation schenla that is decolnposed into two schemas with attribute sets X' and Y, and let F be a set of FDs over R. The **projection of F on X** is the set of FDs in the closure $I^{i'}+$ (not just $.F$!) that involve only attributes in X. We denote the projection of F on attributes X as F_X. Note that a dependency $U \rightarrow V$ in $F+$ is in F_X only if *all* the attributes in [/ and V are in X.

The decornposition of relation scherna R with FI)s F into schcrnas with attribute sets X and Y is dependency-preserving if $(F_X \cup F_Y)^+ = F^+$. That is, if we take the dependencies in F_X and F_Y and cornpute the closure of their un.ion, we get back all dependencies in the closure of F. rrherefore, we need to enforce onl,y the dependencies in F_X and F_Y; allFDs in F^+ are then sure to be satisfied. To enforce F_X ,\vs need to exarnine only relation)((on in.serts to that relation). To enforce F_Y, we need to exarnine only relation Y.

To appreciate the need to consider the closure F^+ while COIUpllting the projection of F, suppose that a relation R with attributes ABC is decomposed into relations\vith attributes AB and $Be:$. The set F of FDs overR includes $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$. Of these, $A \rightarrow B$ is in F_{AB} and $B \rightarrow C$ is in F_{BC}. But is this decoIIlposition dependency-preserving? What about $C \rightarrow A$? This dependency is not irnplied by the dependencies listed (thus far) for F_{AB} and F_{BC}.

The closure of F contains all dependencies in F plus $A \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. Consequently, F_{AB} also contains $B \rightarrow A$, and F_{BC} contains $C \rightarrow B$. Therefore, $F_{AB} \cup F_{BC}$ contains $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. The closure of the dependencies in F_{AB} and F_{BC} now includes $C \rightarrow A$ (which follows frorn $C \rightarrow B$, $B \rightarrow A$, and transitivity). l'hus, the deccHnposition preserves the dependency $C \rightarrow A$.

A direct application of the definition gives us a straightforward algoritlun for testing whether a deconlposition is dependency-preserving. (This algorithrn is exponential in the size of the dependency set. A polynomial algorithnl is available; see Exercise 19.9.)

We began this sectiol1with an exanlple of a lossless-join deC0111position that was not dependency-preserving. Other decorupositions are dependency-preserving, but not lossless. A silnple example consists of a relation ABC' with FD $A \rightarrow B$ that is decornposed into AB and BG.

19.6 NORMALIZATION

Having covered the concepts needed to understand the role of HortHa} forms and decolnpositions in database design, we now consider algoritlnIls for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join deccunpositioll into a collection of BCNF relation schemas. Unfortunately, there may be no dependenc,y-preserving decOlnposition into a collection of BCN.F relation schernas. However, there is always a dependency-preserving, lossless-join decoruposition into a collection of 3NF relation schernas.

19.6.1 Decomposition into BCNF

We now present an algorithm for decornposing a relation scherna R with a set of FI)sF into a collection of BCNF relation schernas:

1. Suppose that R is not in BCNF. Let $X \subset R$, A be a single attribute in R, and $X \rightarrow A$ be an FD that causes a violation of BCNF. DecornposeR into R - A and XA.

2. If either R - A or XA is not in BCN.F, decornpose thern further by a recursive application of this algorithrn.

$R - A$ denotes the set of attributes other than A in R, and XA denotes the union of attributes in X and A. Since $X \rightarrow A$ violates BCNF, it is not a trivial dependency; further, A is a single attribute. Therefore, A is not in X; that is, $X \cap A$ is ernpty. Therefore, each dec()lnposition carried out in Step 1 is lossless-j oin.

The set of dependencies associated with $R - A$ and XA is the projection of F onto their attributes. If one of the new relations is not in BCNF, w'e decornpose it further in Step 2. Since a decornposition results in relations with strictly fewer attributes, this process terrninates, leaving us with a collection of relation schernas that are all in BCNF. Further, joining instances of the (two or 1nore) relations obtained through this algorithrn yields precisely the corresponding instance of the original relation (i.e., the decorllposition into a collection of relations each of which in BCNF is a lossless-join dec()lnposition).

Consider the Contracts relation with attributes $C3JDPQV$ and key C. We are given FDs $JP \rightarrow C$ and $3D \rightarrow P$. By using the dependency $3D \rightarrow P$ to guide the decornposition, we get the two schernas $3DP$ and $C5JDQV$. $51)P$ is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: $.I \rightarrow 5$. This rneans that the sche1na $CSJDQV$ is not in BCNF So we deccnnpose it further into $J3$ and $C.IDC2V$. $C \rightarrow JDQV$ holds over $CJDQV$; the only other FI)s that hold are those obtained frorll this PI) by augrnentation, and therefore all FDs contain a key in the left side. Thus, each of the schernas $ST)P$, $.IS$, and $CJDQV$ is in BCNF, and this collection of schcrnas also represents a lossless-join decornposition of $CSJDQV$.

The steps in this deC(nllposition process can be visualized as a tree, as shown in Figure 19.10. The root is the original relation $CSJIJPQV$, and the leaves are the BCNF relations that result frorn the deccHnposition aJgorithrn: $3D?$, $.IS$, and $CSDQV$. Intuitively, ea.ch internal node is replaced by its children through a single decomposition step guided by the FD shown just below the node.

Redundancy in BCNF Revisited

The decolnposition of $CSJDQV$ iuto SDP, JS, and $CJDQV$ is not dependency-preserving. Intuitively, dependency $Jp \rightarrow C$ carlllot be enforced without a, join. ()ne way to deal \vith this situation is to add a relation \vith attributes $GJ\})$. In

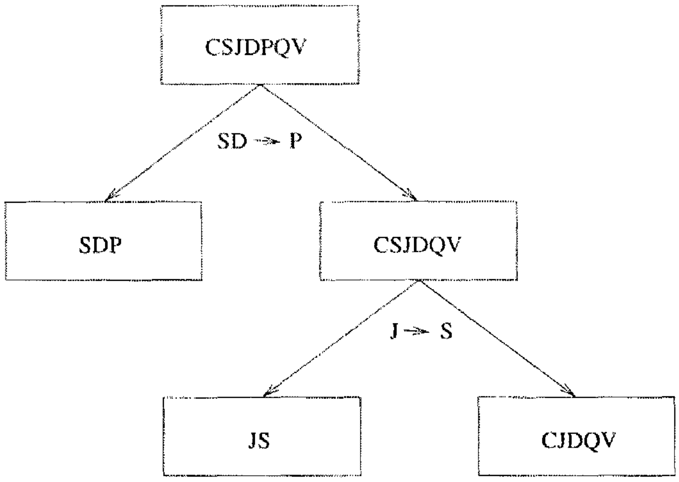


Figure 19.10 Decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV*

effect, this solution arnounts to storing SOITle information redundantly to rnake the dependency enforcement cheaper.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet some redundancy can be predicted by FD infonnation. In particular, if we join the relation instances for *SDP* and *CJDQVand* project the result onto the attributes *CJP*, we rnust get exactly the instance stored in the relation with scherna *CJP*. We saw in Section 19.4.1 that there is no such redundancy within a single BCNF relation. This exarnple shows that redundancy can still occur across relations, even though there is no redundancy within a relation.

Alternatives in Decomposing to BCNF

Suppose several dependencies violate BCNF. Depending on ·which of these dependencies we choose to guide the next decornposition step, we rnay arrive at quite different collections of BeNF relations. Consider Contracts. We just decornposed it into *SDP*, *is*, and *CJDQV*. Suppose we choose to decornpose the original relation *CSJDPQV* into *JS* and *CJDPQV*, based on the FD $.I \to S$. The only dependencies that hold over *CJDPQV* are $.IP \to C$ and the key dependency $C \to C.IDPQV$. Since *iP* is a key, *CJDPQV* is in BeNF. Thus, the schernas *JS* and *CJDPQV* represent a lossless-join decornposition of Contracts into BCNF relations.

The lesson to be learned here is that the theor,Y of dependencies can tell us ·when there is redundancy and give us clues about possible clecornpositions to address the problern, but it cannot discrirninate arnong decornposition alternatives. A designer has to consider the alternatives and choose one based on the scrnantics of the application.

BCNF and Dependency-Preservation

Sometimes, there simply is no decomposition into BCNF that is dependency-preserving. As an example, consider the relation schema SBD, in which a tuple denotes that sailor S has reserved boat B on date D. If we have the FDs $SB \rightarrow D$ (a sailor can reserve a given boat for at most one day) and $D \rightarrow B$ (on any given day at most one boat can be reserved), SBD is not in BCNF because D is not a key. If we try to decompose it, however, we cannot preserve the dependency $SB \rightarrow D$.

19.6.2 Decomposition into 3NF

Clearly, the approach we outlined for lossless-join decomposition into BCNF also gives us a lossless-join decomposition into 3NF. (Typically, we can stop a little earlier if we are satisfied with a collection of 3NF relations.) But this approach does not ensure dependency-preservation.

A simple modification, however, yields a decomposition into 3NF relations that is lossless-join and dependency-preserving. Before we describe this modification, we need to introduce the concept of a minimal cover for a set of FDs.

Minimal Cover for a Set of FDs

A minimal cover for a set F of FDs is a set G of FDs such that:

1. Every dependency in G is of the form $X \rightarrow A$, where A is a single attribute.
2. The closure F^+ is equal to the closure G^+.
3. If we obtain a set H of dependencies from G by deleting one or more dependencies or by deleting attributes from a dependency in G, then $F^+ \neq H^+$.

Intuitively, a minimal cover for a set F of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required for the closure to be equal to F^+.

As an example, let F be the set of dependencies:

$$A \rightarrow B,\ ABCD \rightarrow E,\ EF \rightarrow G,\ EF \rightarrow H, \text{ and } ACDF \rightarrow EG.$$

First, let us rewrite $ACDF \rightarrow EG$ so that every right side is a single attribute:

$ACDF \rightarrow E$ and $ACDF \rightarrow G$,

Next consider $ACDF \rightarrow G$, This dependency is irnplied by the following FDs:

$A \rightarrow B$, $ABCD \rightarrow E$, and $EF \rightarrow G$,

Therefore, we can delete it, Sirnilarly, we can delete $ACDF \rightarrow E$. Next consider $ABCD \rightarrow E$, Since $A \rightarrow B$ holds, we can replace it with $ACD \rightarrow E$, (At this point, the reader should verify that each rernaining FD is rninilnal and required,) Thus, a minimal cover for F is the set:

$A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, and $EF \rightarrow H$,

The preceding exarnple illustrates a general algorithrn for obtaining a rninimal cover of a set F of FDs:

1. Put the FDs in a Standard Form: Obtain a collection G of equivalent FDs with a single attribute on the right side (using the decornposition axiolIl),
2. Minimize the Left Side of Each FD: For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to $F+$,
3. Delete Redundant FDs: Check each reluaining FD in G to see if it can be deleted while preserving equivalence to $.F+$,

Note that the order in which we consider FDs while applying these steps could produce different rninilnal covers; there could be several rninirnal covers for a given set of FDs,

lV101'8 irnportant, it is necessary to rniniInize the left sides of F'Ds *before* checking for redundant FI)s, If these two steps are reversed, the final set of FI)s could still contain senne redundant FDs (i,e., not be a rninirnal cover), as the following exarnple illustrates, LetF be the set of dependencies, each of which is already in the standard fornl:

$ABCD \rightarrow E$, $E \rightarrow D$, $A \rightarrow B$, and $AC \rightarrow$ I),

Observe that none of these FDs is redundant; if we checked for redundantFDs first, we would get the same set of FI)s F. The left side of *il13CIJ* E can be replaced by AC while preserving equivalence to F^+, and we \vould stop here if \ve checked for reclunda.ntF'Ds in F before rnillilnizing the left sides. However, the set of FDs we have is not a Ininlnal cover:

$AC \rightarrow E, E \rightarrow D, A \rightarrow B$, and $AC \rightarrow D$.

From transitivity, the first two FDs irnply the last FD, which can therefore be deleted while preserving equivalence to F^+. The irnportant point to note is that $AC \rightarrow D$ becc)lnes redundant only after we replace $ABeD \rightarrow E$ with $AC \rightarrow E$. If we Ininirnize left sides of FDs first and then cheek for redundantFDs, we are left with the first three FDs in the preeeding list,whieh is indeed a Ininirna1 cover for F.

Dependency-Preserving Decomposition into 3NF

Returning to the problenl of obtaining a lossless-join, dependency-preserving decornposition into 3NF relations, let R be a relation with a set [/' of FDs that is a minirnal cover, and let $R_1, R_2, \ldots, R_n$ be a lossless-join decolnposition of R. For $1 \leq i \leq n$, suppose that each R_i is in 3NF and let F_i denote the projection of F onto the attributes of R_i. Do the following:

- Identify the set N of dependencies in F that is not preserved, that is, not included in the closure of the union of *Fis*.
- :For each FD $X \rightarrow A$ in N, create a relation schelna XA and add it to the decomposition of R.

Obviously, every dependency in F is preserved if we replace R by the R_is plus the schernas of the forn1 XA added in this step. The Ris are given to be in 3NF. We can show that each of the schemas XA is in 3NF as follows: Since $X \rightarrow A$ is in the lninirnal cover F, $Y \rightarrow A$ does not hold for any Y that is a strict subset of X. Therefore, X is a key for XA. :F\llrther, if any other dependencies hold over XA, the right side can involve only attributes in X' because A is a single attribute (because $X \rightarrow A$ is an FD in a rninhnal cover). Since X is a key for ..:YA, none of these additional dependencies causes a violation of 3NF (although they rnight cause a violation of BCNF).

As an optilYlization, if the set N contains several FI)swith the saIne left side, say, $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$, we can replace thern \vith a single equivalent FD $X \rightarrow$ AI $\ldots A_n$. Therefore, we produce one relation scherna $XA_1 \ldots A_n$, instead of several schernas $XA_1, \ldots, XA_n$, \vhich is generally preferable.

Consider the Contracts relation with attrilnltes $CSJDPQV$ and FI)s $JP \rightarrow C$, $SD \rightarrow P$, and $J \rightarrow S$. If we decolnpose $CSJDPQV$ into *SDIJ* and $CSJDQV$, then $8DP$ is in BCNF, but $CSJDQV$ is not even in 3NF. So \ve dec.olupose it further into JS and $CJDQV$. rrhe relation schemas SDP, *.I8*, and $CJDQV$ are in 3NF (in fact, in BCNF), and the decoInposition is lossless-join. However, the dependency $JP \rightarrow C$ is not preserved. This problerIl can be addressed by adding a relation schema CJP to the decornposition.

3NF Synthesis

We assurned that the design process starts with an ER diagraII1, and that our use of FDs is primarily to guide decisions about decolnposition. The algorithill for obtaining a lossless-join, dependency-preserving decornpositiol1 was presented in the previous section fro111 this perspective--------a lossless-join decoru-position into 3NF is straightforward, and the algorithrn addresses dependency-preservation by adding extra relation schcrnas.

An alternative approach, called synthesis, is to take all the attributes over the original relation R and a rnininlal cover F for the FDs that hold over it and add a relation scherna XA to the decomposition of R for each FD $X \rightarrow A$ in F.

The resulting collection of relation schernas is in 3NF and preserves all FDs. If it is not a lossless-join decomposition of R, we can Dlake it so by adding a relation schenla that contains just those attributes that appear in sorne key. This algorithrn gives us a lossless-join, dependency-preserving decornposition into 3NF and has polynornial corIlplexity-----polynornial algorithms are available for coruputing rninirnal covers, and a key can be found in polync)Inial tirHe (even though finding all keys is known to be NP-cornplete). The existence of a polynornial algorithnl for obtaining a lossless-join, dependency-preserving decornposition into 3NF is surprising when we consider that testing whether a given schema is in 3NF is NP-cornplete.

As an exarnple, consider a relation ABC with FI)s $F = \{A \rightarrow B, C \rightarrow B\}$. The first step yields the relation scheluas AB and BG. This is not a lossless-join deC0l11position of $AilC$; $AB \cap BC$ is B, and neither $B \rightarrow A$ nor $B \rightarrow C$ is in F^+. If we add a, schema AC;, we have the lossless-join property as well. Although the collectic)ll of relations AB,BC, and AC is a depenclency-preserving, lossless-join decornposition of ABC, we obtained it through a process of *synthesis*, rather tllan through a process of repeated decornposition. We note that the decoIIIposition produced by the synthesis approa,ch heavily dependends on the rninirnal cover used.

As another example of the synthesis approach, consider the Contracts relation with attributes $CSJDPQV$ and the followving FI)s:

$$C \quad CSJDPQV, .IP \rightarrow C, 8D \rightarrow P, \text{ and } J \rightarrow S.$$

This set of FI)s is not a rninirnal cover, and so we IIIUSt find OI18. We first replace $G \rightarrow CSJDPQV$ with tllcF'I)s:

$C \rightarrow 5$, $C \rightarrow J$, $C \rightarrow IJ$, $C \rightarrow P$, $C \rightarrow Q$, and $C \rightarrow V$.

The FD $C \rightarrow P$ is irnplied by $C \rightarrow S$, $C \rightarrow D$, and $SD \rightarrow P$; so we can delete it. The FD $C \rightarrow S$ is irnplied byC $\rightarrow J$ and $J \rightarrow S$; so we can delete it. This leaves us with a rninirnal cover:

C $\rightarrow$,*I*, C $\rightarrow$ 1), $C \rightarrow Q$, C $\rightarrow V$, $JP \rightarrow$ C, $3D \rightarrow P$, and $J \rightarrow S$.

lJsing the algorithrll for ensuring dependency-preservation, we obtain the relational scherna *CJ*, *CD*, *CQ*, *CV*, *GIJP*, *SDP*, and *JB*. We can irnprove this schenla by cornbining relations for which C is the key into *CDJPQV*. In addition, we have *SDP* and *,IS* in our decorllposition. Since one of these relations (*CDJPQ*V) is a superkey, we are done.

Conlparing this decomposition with that obtained earlier in this section, we find they are quite close, with the only difference being that one of them has *CDJPQV* instead of *CJP* and *CJDQV*. In general, however, there could be significant differences.

19.7 SCHEMA REFINEMENT IN DATABASE DESIGN

We have seen how normalization can eliminate redundancy and discussed several approaches to nonnalizing a relation. We now consider how these ideas are applied in practice.

Database designers typically use a conceptual design rnethodology, such as ER design, to arrive at an initial database design. Given this, the approach of repeated decorllpositions to rectify instances of redundancy is likely to be the rnost natural use of PI)s and nonnalization techniques.

In this section, we Inotivate the need for a schcrna refinernent step followving ER design. It is natural to ask whether we even need to decornpose relations produced by translating an ER diagranl. Should a good ER design not lead to a collection of relations free of redundancy prob.lerns? Unfortunately, ER design is a c()!nplex, subjective process, and certain constraints are not expressible in tenns of ER diagraJns. The exaruples in this section are intendecl to illustrate why decornposition of relations produced through ER design rnight be necessary.

19.7.1 Constraints on an Entity Set

Consider the Hourly_Emps relation again. rrhe constraint that attribute *ssn* is a key can be expressed as an FI):

$$\{ssn\} \rightarrow \{ssn, name, lot, rating, hourly\_wages, hours\_worked\}$$

:For brevity, we \vrite this FD as $S \rightarrow SNLRWH$, using a single letter to denote each attribute and ornitting the set braces, but the reader should rernernber that both sides of an FD contain sets of attributes. In addition, the constraint that the *hourly_wages* attribute is deterruined by the *rating* attribute is an FD: $R \rightarrow W$.

As we saw in Section 19.1.1, this FI) led to redundant storage of rating wage associations. *It cannot be expressed in terms of the ER model. Only FDs that determine all attributes of a relation (i.e., key constraints) can be expressed in the ER rnodel.* rrherefore, we could not detect it when we considered Hourly_EIllPS as an entity set during ER IIlodeling.

We could argue that the problenl with the original design was an artifact of a poor ER design, which could have been avoided by introducing an entity set called Wage_Table (with attributes *rating* and *hourly_wages*) and a relationship set lIas_Wages associating IIourly_ErIlps and Wage_Table. The point, however, is that we could easily arrive at the original design given the subjective nature of ER rnodeling. Having forInal techniques to identify the problenl with this design and guide us to a, better design is very useful. The value of such techniques cannot be underestirnated when designing large schernas·····schcrnas with rnore than a hundred tables are not unCOIlllIHon.

19.7.2 Constraints on a Relationship Set

The previous exarnple illustrated how FDs can help to refine the subjective decisions Blade during ER. design, but one could argue that the best possible ER, eliagrarn \vould have led to the same final set of relations. ()ur next exarnple shows how Ff) inforrnation ca.ll lead to a. set of relations unlikely to be arrived at solely through ER design.

We revisit an example froTH Chapter 2. Suppose that we have entity sets Parts, Suppliers, and I)epartrnents, as \vell as a relationship set Contracts that involves all of theIn. We refer to the scherna for Contra(:ts as *CQPSD*. A contra,ct with contract id C specifies that a supplier S will supply sorne quantity Q of a part P to a departrnent D. (We have adderl the contract ieI field C' tC) tIle versiorl of the Contracts relation discussed in Chapter 2.)

We Blight have a policy that a departrnent purchases at Inost one part fror11 any given supplier. Therefore, if there are several contracts between the saIne supplier and departrnent, \ve know that the saIne part Inus!; be involved in all of thcrn. This constraint is an FD, $DS \rightarrow P$.

Again we have redundancy and its associated problclns. We can address this situation by decornposing Contracts into two relations with attributes *CQSD* and *3DP*. Intuitively, the relation *3DP* records the part supplied to a depart-rllent by a supplier, and the relation *C:QSD* records additional infornlation about a contract. It is unlikely that we would arrive at such a design solely through ER rIlodeling, sinee it is hard to fOfrnulate an entity or relationship that corresponds naturally to *CQSD*.

19.7.3 Identifying Attributes of Entities

This exarIlple illustrates how a careful examination of FDs can lead to a better understanding of the entities and relationships underlying the relational tables; in particular, it shows that attributes can easily be associated with the 'wrong' entity set during ER design. The ER diagrarn in Figure 19.11 shows a relationship set called Works_In that is similar to the Works.ln relationship set of Chapter 2 but with an additional key constraint indicating that an employee can work in at rnost one departrIlent. (Observe the arrow connecting Employees to Works_In.)

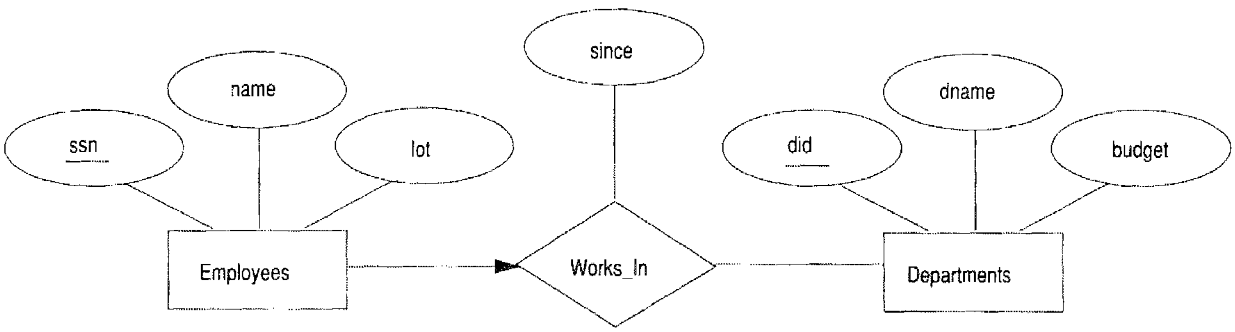


Figure 19.11. The Works._In Relationship Set

Using the key constraint, we can translate this ER diagrarn into two relations:

Workers(*ssn*, *narne*, *lot*, *d'id*, *since*)
Departrnents(*did*, *dname*, *budget*)

The entity set Ernployees and the relationship set Works_In are rnapped to a single relation, vVorkers. This translation is based on the second approach discussed in Section 2.4.1.

Now suppose elIlployees are assigned parking lots based on their departrnent, and that all enlployees in a given departrnent are assigned to the saIne lot. This constraint is not expressible with respect to the ER, diagrarIl of Figure 19.1l. It is another example of an FD: $did \rightarrow lot$. The redundancy in this design can be elirninated by decornposing the Workers relation into two relations:

vVorkers2(*ssn*, *name*, *did*, *since*)
Dept_Lots(*did*, *lot*)

'rhe new design has lnuch to reconunend it. We can change the lots associated with a departlnent by updating a single tuple in the second relation (i.e., no update anornalies). We can associate a lot with a department even if it currently has no crnployees, without using *null* values (i.e., no deletion anornalies). We can add an eruployee to a department by inserting a tuple to the first relation even if there is no lot associated with the enlployee's departrnent (i.e., no insertion anornalies).

Exalnining the two relations Departrnents and Dept_Lots, which have the saIne key, we realize that a Departrnents tuple and a Dept_Lots tuple with the sarne key value describe the sarne entity. This observation is reflected in the ER cliagrarn shown in Figure 19.12.

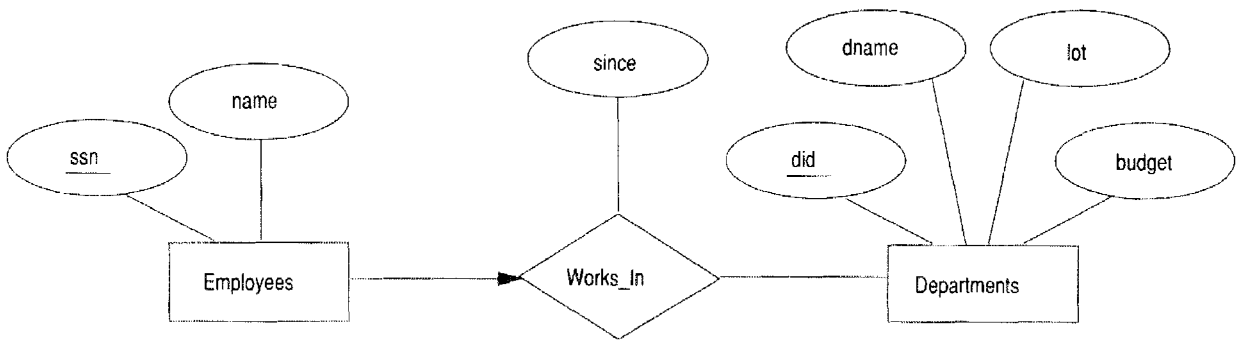


Figure 19.12 Refined\Norks_In Relationship Set

Translating this diagrarn into the relational rnodel would yield:

Workers2(*8871"* *narne*, *did*, *since*)
I)epartrnentsCdid, *dname*, *budget*, *lot*)

It SeClllS intuitive to associate lots with crnployees; on the other hand, the les reveal tllat ,in this exarnple lots are really associated with departrnents. The subjective process of ER modeling could Iniss this point. T'he rigorous process of norrnaliza,tion would not.

19.7.4 Identifying Entity Sets

Consider a variant of the Reserves scherna used in earlier chapters. Let Reserves contain attributes S, B, and D as before, indicating that sailor S has a reservation for boat B on day D. In addition, let there be an attribute C denoting the credit card to which the reservation is charged. We use this exarnple to illustrate how FD illfonnation can be used to refine an ER design. In particular, we discuss how FD inforluation can help decide whether a concept should be rnodeled as an entity or as an attribute.

Suppose every sailor uses a unique credit card for reservations. This constraint is expressed by the FD $S \rightarrow C$. This constraint indicates that, in relation Reserves, we store the credit card rnllnber for a sailor as often as we have reservations for that sailor, and we have redundancy and potential update anolnalies. A solution is to deconlpose Reserves into two relations with attributes *SBD* and *SC*. Intuitively, one holds infornnation about reservations, and the other holds infonnation about credit cards.

It is instructive to think about an ER design that would lead to these relations. One approach is to introduce an entity set called Credit_Cards, with the sale attribute *cardno*, and a relationship set Has_Card associating Sailors and Credit_Cards. By noting that each credit card belongs to a single sailor, we can Inap Has_Card and Credit_Cards to a single relation with attributes *SC*. We would probably not rnodel credit card nUlnbers as entities if our Inain interest in card nurnbers is to indicate how a reservation is to be paid for; it suffices to use an attribute to rnodel card nUlnbers in this situation.

A second approach is to rnake *cardno* an attribute of Sailors. But this approach is not very natural-.–a sailor lllay have several cards, and we are not interested in all of theln. Our interest is in the one card that is used to pay for reservations, which is best lnodeled as an attribute of the relationship Reserves.

A helpful way to think about the design problern in this exarnple is that we first lnake *cardno* an attribute of H,eserves and then refine the resulting tables by taking into account the FD information. (Whether we refine the design by adding *cardno* to the table obtained froTll Sailors or by creating a new table with attributes *SC* is (1, separate issue.)

19.8 OTHER KINDS OF DEPENDENCIES

FI)s are probal)l.y the rn08t conunon and irnportant kind of constraint froIII the point of view of database design. However, there are several other kinds of dependencies. In particular, there is a well-developed theory for database design llsing *multivalued dependencies* and *join dependencies*. By taking sueh dependencies into account, we can identify potential redundancy problenls that cannot be detected using FDs alone.

'rhis section illustrates the kinds of redundancy that can be detected using II1Ul-tivalued dependencies. Our Inain observation, however, is that sirnple guidelines (which can be checked using only FD reasoning) can tell us whether we even need to worry about complex constraints such as 111ultivalued and join dependencies. We also conunent on the role of *inclusion dependencies* in database design.

19.8.1 Multivalued Dependencies

Suppose that we have a relation with attributes *course, teacher,* and *book,* which we denote as *CTB*. The Ineaning of a tuple is that teacher *T* can teach course *C,* and book *B* is a reccnnmended text for the course. There are no FDs; the key is *CTB*. However, the recolnlnended texts for a course are independent of the instructor. The instance shown in Figure 19.13 illustrates this situation.

course	*teacher*	*book*
Physics101	Green	Mechanics
PhysicslOl	Green	Optics
PhysicslOl	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

Figure 19.13 BCNF R.elation with Redundancy That Is Revealed by MVDs

Note three points here:

- The relation sehcrna *CTB* is in BCNF; therefore we would not consider decolnposing it further if we looked only at the FDs that hold over *(JTB.*
- There is redundancy. rrhe fact that G·reen can teach Physics101 is recorded once per recommended text for the course. Sirnila.rly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- T'he redundancy can be elirninated by decornposing *CTB* into CT and *CE.*

The redundaJ1cy in this exarnple is due to the constraint that the texts for a course are independent of tIle instructors, which cannot be expressed in tenns of FDs. rrhis constraint is an example of a *multivalued dependency,* or MVD. Ideally, we should rnodel this situation using two binary relationship sets, Instructors with attributes CT and Text with attributes CB. Because these are two essentially independent relationships, rnodeling them with a single ternary relationship set with attributes CTE is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, ho\vever, we rnight create a ternary relationship. A careful analysis of the fvIVD infonnation would then reveal the problern.

Let R be a relation scheIna and let X and Y be subsets of the attributes of R. Intuitively, the **multivalued dependency** $X \rightarrow\rightarrow$ Y'is said to hold over R if, in every legal instance r of R, each X value is associated with a set of Yvalues and this set is independent of the values in the other attributes.

ForInally, if the MVD $X \rightarrow\rightarrow Y$ holds over R and $Z = R - XY$, the following lllUSt be true for every legal instance r of R:

> If tl E r, $t2$ E rand $tl.X = t2.X$, then there must be sorne $t3$ E r such that $tl'XY = t3.XY$ and $t2 \cdot Z = t3'Z$,

Figure 19.14 illustrates this definition. If we are given the first two tuples and told that the MVD $X \rightarrow\rightarrow Y$ holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the first two tuples—treating the first tuple as $t2$ and the second tuple as t_1—we can deduce that the tuple $t4$ must also be in the relation instance.

LX	Y	Z]	
a	b_1	CI	— tuple $t1$
a	b_2	C2	— tuple $t2$
a	b]	C2	— tuple t_3
a	$b2$	CI	— tuple $t4$

Figure 19.14 Illustration of MVD Definition

This table suggests another way to think about lVIVDs: If $X \rightarrow\rightarrow Y$ holds over R, then $\pi_{YZ}(\sigma_{X=x}(R)) = \pi_Y(\sigma_{X=x}(R)) \times \pi_Z(\sigma_{X=x}(R))$ in every legal instance of R, for any value x that appears in the X colurnn of R. In other words, consider groups of tuples inR with the sarne X-value. In each such group consider the projection onto the attributes YZ. This projection HUlst be equal to the cross-product of the projectiolls onto Yand Z. That is, for a given X-value, the Y-values and Z-values are independent. (Froln this definition it is easy to see that $X \rightarrow\rightarrow$ y'l11Ust hold wherlever $X \rightarrow Y$ holds. If the FI) $X \rightarrow$

Y holds, there is exactly one Y-value for a given X-value, and the conditions in the MVD definition hold trivially. The converse does not hold, as Figure 19.14 illustrates.)

Returning to our CTB exaIllple, the constraint that course texts are independent of instructors can be expressed as $C \rightarrow\rightarrow T$. In terlllS of the definition of MVDs, this constraint can be read as follo\vs:

> If (there is a tuple showing that) C is taught by teacher T,
> and (there is a tuple showing that) G has book B as text,
> then (there is a tuple showing that) G is taught by T and has text B.

Given a set of FDs and MVDs, in general, we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three ArIllstrong AxioIllS plus five additional rules. Three of the additional rules involve only MVDs:

- MVD Complementation: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$.
- MVD .Augmentation: If $X \rightarrow\rightarrow$ Yand $W \supseteq Z$, then $WX \rightarrow\rightarrow YZ$.
- MVD Transitivity: If $X \rightarrow\rightarrow$ Yand $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow (Z - Y)$.

As an exanlple of the use of these rules, since we have $C \rightarrow\rightarrow T$ over GTB, MVD complelnentation allows us to infer that $C \rightarrow\rightarrow OTB - CT$ as well, that IS, $C \rightarrow\rightarrow B$. The remaining two rules relate FDs and MVDs:

- Replication: If $X \rightarrow$ Y, then $X \rightarrow\rightarrow Y$.
- Coalescence: If $X \rightarrow\rightarrow$ Yand there is a W such that W n Y- is elnpty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

()bserve that replication states that every FD is also an MVD.

19.8.2 Fourth Normal Form

Fourth Horrnal fonn is a direct generalization of BeNF. Let R be a relation scherna, X and Y be nonernpty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in **fourth normal form (4NF)**, if, for every l\1.VI) $X \rightarrow\rightarrow Y$ that holds over R, one of the following staternents is true:

- $Y \subseteq X$ or $XY = R$, or
- X is a superkey.

In reading this definition, it is irnportant to understand that the deflnition of a *key* has not changed··········-the key rnust uniquely deterrnine all attributes through FDs alone. $X \rightarrow\rightarrow$ Y'is a trivial MVD if $Y \subset X \subseteq R$ or $XY = R$; such MVDs always hold.

The relation *CTB* is not in 4NF because $C \rightarrow\rightarrow T$ is a nontrivial MVD and C is not a key. We can elirninate the resulting redundancy by deconlposing *CTB* into *Cr* and *CB;* each of these relations is then in 4NF.

To use MVD inforrnation fully, we nUlst understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions-detected using only FD information!—under which we can safely ignore MVD inforrnation. That is, using MVD information in addition to the FD infornlation will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

> If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An in1.portant assl.unption is inlplicit in any application of the preceding result: *The set of FDs identified thus far is 'indeed the set of all FDs that hold over the relation.* This assulllption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

We illustrate this point using an exalnple. Consider a relation scherna *ABCD* and suppose that the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ are given. Considering only these dependencies, this relation schema appears to be a counterexalnple to the result. The relation has a sirnple key, appears to be in BCNF, and yet is not in 4NF because $B \rightarrow\rightarrow C$: causes a violation of the 4NF conditions. Let us take a closer look.

B	C	A	D	
b	Cl	0:1	d_1	— tuple t_1
b	C2	([,2	d_2	— tuple t_2
b	Cl	([,2	d2	— tuple t_3

Figure 19.15 Three Tuples [rorn a Legal Instance of *ABCD*

Figure 19.15 8ho\v8 three tuples f1'om an instance of *ABCD* that satisfies the given MVD $B \rightarrow\rightarrow C$. Frolu the definition of an MVD, given tuples tl and t_2, it follows that tuple t_3 Inust also be included in the instaJ1ce. Consider tuples t_2 and t_3. FrOIJI the givenFD $A \rightarrow BCD$ and the fact that these tuples have the same A-value, we can deduce that $c_1 = c_2$. Therefore, we see that the FD $B \rightarrow$ C rnust hold over $ABCD$ whenever the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ hold. If $B \rightarrow C$ holds, the relation $ABeD$ is not in BeNF (unless additional FDs lllake B a key)!

Thus, the apparent counterexalnple is really not a counterexalllple··········-rather, it illustrates the iInportance of correctly identifying all FDs that hold over a relation. In this exarnple, $A \rightarrow BCI)$ is not the only FD; the FD $B \rightarrow C$ also holds but ·was not identified initially. Given a set of FDs and IvIVI)s, the inference rules can be used to infer additional FDs (and l\1VDs); to apply the Date-Fagin result without first using the l\1VD inference rules, we IUUSt be certain that we have identified all the FDs.

In summary, the Date-Fagin result offers a convenient way to check that a relation is in 4NF (without reasoning about l\1VDs) if we are confident that we have identified all FDs. At this point, the reader is invited to go over the examples we have discussed in this chapter and see if there is a relation that is not in 4NF.

19.8.3 Join Dependencies

A join dependency is a further generalization of MVDs. A join dependency (JD) $\bowtie \{R_1, \ldots, R_n\}$ is said to hold over a relation R if $R_1, \ldots, R_n$ is a lossless-join decolnposition of R.

An MVD $X \rightarrow\rightarrow Y$over a relation R can be expressed as the join dependency $\bowtie$ {XV, X(R,--Y)}..As an example, in the GTB relation, the MVD $C \rightarrow\rightarrow T$ can be expressed as the join dependency $\bowtie$ {Crr, CB}.

U·nlike FDs and l'v1VDs, there is no set of sound and cornplete inference rules for JDs.

19.8.4 Fifth Normal Form

A relation schcrna R is said to be in fifth normal form (5NF) if, for every .II) $\bowtie \{R_1, \ldots, R_n\}$ that holds over R, one of the follo"ving statcrnents is true:

- $R_i = R$, for scnne i, or
- The .lD is irnplied by the set of those FDs over R in ·which the left side is a key for R.

The second condition deserves s(Hne explanation, since we have not presented inference rules for FDs and .00Ds taken together. Intuitively, we rnust be able to sho\v that the decolnposition of R into $\{R_1, \ldots, R_n\}$ is lossless-join whenever the key dependencies (FDs in which the left side is a key for *R)* hold. JI) $\bowtie \{R_1, \ldots, R_n\}$ is a trivial JD if $R_i = R$ for SaIne i; such a .JD always holds.

The following result, also due to Date and Fagin, identifies conditions— again, detected llsing only FD inforlnation---under -which we can safely ignore JD inforlnation:

> If a relation schenla is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF but not necessary. rrhe result can be very useful in practice because it allows us to conclude that a relation is in 5NF *'Without ever 'identifying the MVDs and JDs that 'may hold oveT the relation.*

19.8.5 Inclusion Dependencies

lVIVDs and JDs can be used to guide database design, as we have seen, although they are less COllUllon than FDs and harder to recognize and reason about. In contrast, inclusion dependencies are very intuitive and quite cornrnon. IIowever, they typically have little influence on database design (beyond the ER design stage).

Infonnally, an inclusion dependency is a statement of the fOlTH that s0111e cohunns of a relation are contained in other cohunns (usually of a second relation). A foreign key constraint is an example of an inclusion dependency; the referring colurnn(s) in one relation rnust be contain.ecl in the prirnary key cohnnn(s) of the referenced relation. As another exarnple, if!? and S are two relations obtained 1)y translating two entity sets that every R entity is also an S erltity, we would have an inclusion dependency; projecting R on its key attributes yields a relation contained in the relation obtained by projecting S on its key attributes.

The rnain point to bear in rnind is that we should not split groups of attributes that participate in an inclusion dependency. For exarnple, if we have an inclusion dependency $AB \subseteq$ *Of),* \vhile decornposing the relation scherna containing AB, we should ensure that at least one of the schemas obtained in the deccnnposition contains botJ1 A and B. Otherwise, we cannot check the inclusion clependency $AB \subseteq CD$ without reconstructing the relation containing A13.

Ivlost inclusion dependencies in practice are *key-based*, that is, involve only keys. Foreign key constraints are a good exalnple of key-based inclusion dependencies. An ER diagram that involves ISA hierarchies (see Section 2.4.4) also leads to key-based inclusion dependencies. If all inclusion dependencies are key-based, we rarely have to worry about splitting attribute gTOUps that participate in inclusion dependencies, since decornpositions usually do not split the primary key. N'ote, however, that going fn:)111 3NF to BCNF always involves splitting sOlne key (ideally not the prirnary key!), since the dependency guiding the split is of the fornl $X \rightarrow A$ where A is part of a key.

19.9 CASE STUDY: THE INTERN'ET SHOP

R,ecall froIn Section 3.8 that DBDudes settled on the following scherna:

Books(*isbn:* CHAR(10), *title:* CHAR(8) , *author:* CHAR(80) ,
qty_in_stock: INTEGER, *price:* REAL, *year_published:* INTEGER)
Custolllers(*cid:* INTEGER, *cnaTne:* CHAR(80) , *address:* CHAR(200))
Orders(*orde.rnum,:* INTEGER, *isbn:* CHAR(.10), *cid:* INTEGER,
cardnu'm: CHAR(16), *qty:* INTEGER, *ordeT_date:* DATE, *ship_date:* DATE)

DBDudes analyzes the set of relations for possible redundancy. The Books relation has only one key, *(isbn)*, and no other functional dependencies hold over the table. Thus, Books is in BCNF. The Custorners relation also has only one key, (*cid*), and no other functional depedencies hold over the table. T'hus, Custorners is also in BCNF.

DBI)udes has already identified the pair ⟨*ordernum, isbn*⟩ as the key for the Orders table. In addition, since each order is placed by one custorner on one specific date with one specific credit card nurnber, the following three functional dependencies hold:

$$ordernum \rightarrow cid,\ ordernum \rightarrow order\_date, \text{ and } ordernum \rightarrow cardnum$$

The experts at DBDudes conclude that Orders is not even in 3NF. (Can you see why?) They decide to clecornpose ()rders into the following two relations:

Orders(*ordernum*, *cid*, *order_date*, *cardnum*, a.nd
()rderlists(*ordernum, isbn*, *qty*, *ship_date*)

The resulting t\vo relations, ()rders and ()rderlists, are both in BCNF', and the decornposition is lossless-join since *ordernum* is a key for (the new) ()rders. The reader is invited to check that this decolnposition is also dependency-preserving. For cornpleteness, we give th,e SQL DIJL for the ()rders and Orderlists relations below:

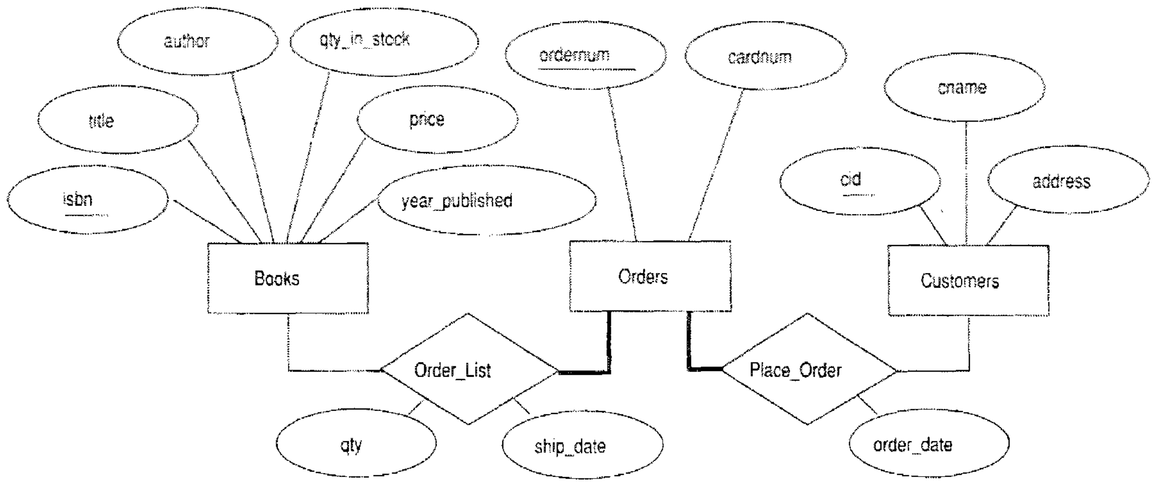


Figure 19.16 ER Diagram Reflecting the Final Design

```
CREATE TABLE Orders ( ordernurn    INTEGER,
                      cid          INTEGER,
                      order_date   DATE,
                      cardnum      CHAR(16),
                      PRIMARY KEY (ordernlllll),
                      FOREIGN KEY (cid) REFERENCES Custolllers )

CREATE TABLE Orderlists (ordernurll  INTEGER,
                         isbn        CHAR(10),
                         qty         INTEGER,
                         ship_date   DATE,
                         PRIMARY KEY (ordernurn, isbn),
                         FOREIGN KEY (isbn) REFERENCES Books)
```

F'igure 19.16 shows an updated ER diagrarn that reflects the new design. Note that DBDudes could have arrived inunedia,tely at this diagrarn if they had made ()rders an entity set instead of a relationship set right at the beginning. But at that tilne they did not understand the requirernents cornpletely, and it seeTHed natural to rnodel Orders as a relationship set. This iterative refinernent process is typical of real-life da,tabase design processes. As DBI)udes has learned over tirne, it is rare to achieve an initial design that is not changed as a project progresses.

T'he DBI)udes team celebrates the successful cornpletion of logical database design and scherna refinelnent by opening a bottle of charnpagne and charging it to B&:N. After recovering frorn the celebration, they lIlove on to the physical design phase.

19.10 REVIEW QUESTIONS

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

- Illustrate redundancy and the problerns that it ca.n cause. Give examples of *insert, delete,* and *update* anoInalies. Can *null* values help address these problerlls? Are they a cOlnplete solution? (Section 19.1.1)
- What is a *decoTnpositio'n* and how does it address redundancy? What problerlls Inay be caused by the use of decolupositions? (Sections 19.1.2 and 19.1.3)
- Define *functional dependencies.* How are *prımary keys* related to FDs? (Section 19.2)
- When is an PD *j implied by* a set *F* of FDs? Define *Armstrong's Axioms,* and explain the statement that "they are a sound and cornplete set of rules for FD inference." (Section 19.3)
- What is the *dependency closure* $F+$ of a set *F* of FDs? What is the *attribute closure* $X+$ of a set of attributes *X* with respect to a set of FDs *F?* (Section 19.3)
- Define INF, 2NF, 3NF, and BCNF. What is the nlotivation for putting a relation in BCNF? What is the motivation for 3NF? (Section 19.4)
- When is the decomposition of a relation schenla *R* into two relation schemas *X* and *Y* said to be a *lossless-join* decomposition? Why is this property so irnportant? Give a necessary and sufficient condition to test whether a decc)1nposition is lossless-join. (Section 19.5.1)
- When is a decornposition said to be *depc'ndency-preserving?* Why is this property useful? (Section 19.5.2)
- Describe how we can obtain a lossless-join decornposition of a relation into BCNF. Give an exanlple to show that there rnay not be a dependency-preserving decornposition into BCNF. Illustrate how a given relation could be decornposed in different ways to arrive at several alternative decornpositions, and discuss the irnplications for database design. (Section 19.6.1)
- Give an example that illustrates how a collection of relations ın BCNF could have redundancy even though each relation, by itself, is free fronl redundancy. (Section 19.6.1)
- What is a. *Tninirnal cover* for a set of FDs? Describe an algorithrn for cornputing the minimal cover of a set of FI)s, and illustrate it with an exarnple. (Section 19.6.2)

- Describe how the algorithl11 for lossless-join decolnposition into BCNF can be adapted to obtain a lossless-join, dependency-preserving decornposition into 3NF. Describe the alternative *synthesis* approach to obtaining such a decorllposition into 3NF. Illustrate both approaches using an exarnple. (Section 19.6.2)
- Discuss how scherna refinernent through dependency analysis and norrnalization can iInprove schemas obtained through ER design. (Section 19.7)
- Define *multivalued dependencies, join dependencies,* and *inclusion dependencies.* Discuss the use of such dependencies for database design. Define 4NF and 5NF, and explain how they prevent certain kinds of redundancy that BCNF does not eliminate. Describe tests for 4NF and 5NF that use only FDs. What key assumption is involved in these tests? (Section 19.8)

EXERCISES

Exercise 19.1 Briefly answer the following questions:

1. Define the term *functional dependency.*
2. Why are some functional dependencies called *trivial*?
3. Give a set. of FDs for the relation schema *R(A,B, C,Dj* with prilnary key *AB* under which *R* is in 1NF but not in 2NF.
4. Give a set of FDs for the relation schelna *R(A,B, C,Dj* with prilnary key *AB* under which *R* is in 2NF but not in 3NF.
5. Consider the relation schelna *R(A,B, OJ,* which has the FD $B \rightarrow C$. If *A* is a candidate key for *R,* is it possible for *R* to be in BCNF? If so, under what conditions? If not, explain why not.
6. Suppose we have a relation schema *R(A, B, OJ* representing a relationship between two entity sets with keys *A* and *13,* respectively, and suppose that *R* has (aIllong others) the FDs *A* $\rightarrow$ Band *13* $\rightarrow$ *A.* Explain what such a pair of dependencies means (i.e., what they irnply about the relationship that the relation nlOdels).

Exercise 19.2 Consider a relation *R* with five attributes *ABCDE.* You are given the follc)\ving dependencies: $A \rightarrow B$, $Be \rightarrow E$, and $ED \rightarrow A$.

1. List all keys for *R.*
2. Is *R* in 3NF?
3. Is *R* in BCNF?

Exercise 19.3 Consider the relation shown in Figure 19.17.

1. List all the functional dependencies that this relation instance satisfies.
2. Assume that the value of attribute *Z* of the last record in the relation is changed frorH z_3 to z_2. Now list all the functional dependencies that this relation instance satisfies.

Exercise 19.4 Assurne that you are given a relation with attributes *ABCD.*

X	Y	Z
Xl	y_1	Zl
Xl	Yl	Z2
X2	Yl	Zl
x_2	Yl	z_3

Figure 19.17 Relation for Exercise 19.3.

1. Asslune that no record has NULL values. \Nrite an SQL query that checks whether the functional dependency $A \rightarrow B$ holds.
2. Assulne again that no record has NULL values. Write an SQL assertion that enforces the functional dependency $A \rightarrow B$.
3. Let us now aSSUlne that records could have NULL values. Repeat the previous two questions under this assurnption.

Exercise 19.5 Consider the following collection of relations and dependencies. Assume that each relation is obtained through decomposition from a relation with attributes *ABCDEFGHI* and that all the known dependencies over relation *ABCDEFGHI* are listed for each question. (The questions are independent of each other, obviously, since the given dependencies over *ABCDEFGHI* are different.) For each (sub)relation: (a) State the strongest nonnal fonn that the relation is in. (b) If it is not in BCNF, decompose it into a collection of BCNF relations.

1. $Rl(A.C,B.D,E)$, $A \rightarrow 13$, $C \rightarrow D$
2. $R2(A,B,F)$, $AC \rightarrow B$, $B \rightarrow F$
3. $R3(A,D)$, $D \rightarrow G$, $G \rightarrow H$
4. $R4(D,C,H,G)$, $A \rightarrow I$, $I \rightarrow A$
5. $R5(A.I,C.B)$

Exercise 19.6 Suppose that we have the following three tuples in a legal instance of a relation schema S with three attributes ABC (listed in order): (1,2,:3), (4,2,3), and (5,3,3).

1. \Which of the following dependencies can you infer does *not* hold over scherna S?

   (a) $A \rightarrow 13$, (b) $Be \rightarrow A$, (c) $13 \rightarrow C$

2. Can you identify allY dependencies that hold over S?

Exercise 19.7 Suppose you are given a relation R with four attributes $ABCD$. For each of the following sets of FDs, assurning those are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) Identify the best Honnal forBl that R satisfies (1NF, 2NF, 3NF, or BeNF). (c) If R is not in BCNF, decOlnpose it into a set of BCNF relations that preserve the dependencies.

1. $C \rightarrow D$, $C \rightarrow A$. 13 C
2. $B \rightarrow C$. $D \rightarrow A$
3. $ABC \rightarrow D$, $D \rightarrow A$
4. $A \rightarrow B$. BC $\rightarrow D$. $A \rightarrow C$
5. $A13 \rightarrow C$, $AB \rightarrow D$. $C \rightarrow A$, $D \rightarrow 13$

Exercise 19.8 Consider the attribute set $R = ABCDEGH$ and the FD set $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, Be \rightarrow A, B \rightarrow G\}$.

1. For each of the following attribute sets, do the following: Cornpute the set of dependencies that hold over the set and write down a rninirnal cover. (ii) Narne the strongest nonnal [onn that is not violated by the relation containing these attributes. (iii) DeCOlnpose it into a collection of BCNF relations if it is IH)t in BeNF'.

   (a) ABC, (b) $ABCD$, (c) $ABCEG$, (d) $DC{:}BGII$, (e) $ACEH$

2. Which of the following decOIllpositions of R = $ABCDEG$, with the saIne set of dependencies F, is (a) dependency-preserving? (b) lossless-join?

   (a) $\{AB, BC, ABDE. EG\}$

   (b) $\{ABC, ACDE, ADG\}$

Exercise 19.9 Let R be decOIllposed into $R_1, R_2, \ldots, R_n$. Let F be a set of FDs on R.

1. Define what it rlleans for F to *be pre8erved* in the set of decOlllposed relations.
2. Describe a polynomial-tirne algorithm to test dependency-preservation.
3. Projecting the FDs stated over a set of attributes X onto a subset of attributes Y requires that we consider the closure of the FDs. Give an exarnple where considering the closure is irnportant in testing dependency-preservation, that is, considering just the given FDs gives incorrect results.

Exercise 19.10 Suppose you are given a relation $R(A,B,C,D)$. For each of the following sets of FDs, assuming they are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) State whether or not the proposed decOlnposition of R into smaller relations is a good decolllposition and briefly explain why or why not.

1. $B \rightarrow C$, $D \rightarrow A$; decornpose into BC and AD.
2. $AB \rightarrow C$, $C \rightarrow A$, $C \rightarrow D$; decompose into ACD and Be.
3. $A \rightarrow BC$, $C \rightarrow AD$; decornpose into ABC and AD.
4. $A \rightarrow B$, B C, $C \rightarrow D$; decornpose into AB and ACD.
5. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decOInpose into AB, AD and CD.

Exercise 19.11 Consider a relation R that has three attributes ABC. It is decornposed into relations R_1 with attributes AB and R_2 with attributes Be.

1. State the definition of a lossless-join decOlnposition with respect to this exarnple. Answer this question concisely by writing a relational algebra equation involving R, R_1, and $H2$.
2. Suppose that $B \rightarrow\rightarrow$ C. Is the decorHposition of R into R_1 and $R2$ lossless-join? Reconcile your answer with the observation that neither of the FDs HI nR2 $\rightarrow$ R_I nor R_I n $R_2 \rightarrow R_2$ hold, in light of the simple test offering **a** necessary and sufficient condition for lossless-join decmnposition into two relations in Section 15.6.1.
3. If you are given the following justa.nees of R_1 a,nd R_2, what can you say about the instance of R from which these were obtained? Answer this question by listing tuples that are definitely ill R and tuples that are possibly in R.

   Instance of R_1 = {(5,l), (6,l)}
   Instance of R_2 = {(l,8), (1,9)}

   Can you say that attribute B definitely *is* or *is not* a key for R?

Exercise 19.12 Suppose that we have the following four tuples in a relation S with three attributes ABC: (1,2,3), (4,2,3), (5,3,3), (5,3,4). Which of the following functional ($\rightarrow$) and rIlultivalued ($\rightarrow\rightarrow$ dependencies can you infer does *not* hold over relation S?

1. $A \rightarrow 13$
2. $A \rightarrow\rightarrow B$
3. $Be \rightarrow A$
4. $BG \rightarrow\rightarrow A$
5. $8 \rightarrow C$
6. $B \rightarrow\rightarrow C$

Exercise 19.13 Consider a relation R with five attributes $ABCDE$.

1. For each of the following instances of R, state whether it violates (a) theFD $Be \rightarrow D$ and (b) the MVD $Be \rightarrow\rightarrow D$:
   (a) { } (i.e., mnpty relation)
   (b) {(0,,2,3,4,5), (2,a,3,5,5)}
   (c) {(o,,2,3,4,5), (2,0,,3,5,5), (o,,2,3,4,6)}
   (d) {(a,2,3,4,5), (2,0,,3,4,5), (o,,2,3,6,5)}
   (e) {(0,,2,3,4,5), (2,0,,3,7,5), (a,2,3,4,6)}
   (f) {(o,,2,3,4,5), (2,0,,3,4,5), (0,,2,3,6,5), (o,,2,3,6,6)}
   (g) {(a,2,3,4,5), (0,,2,:3,6,5), (0,,2,3,6,6), (o,,2,3,4,6)}
2. If each instance for R listed above is legal, what can you say about the FD $A \rightarrow B$?

Exercise 19.14 JDs are lllotivated by the fact that sornetilnes a relation that cannot be decoruposed into two sinaller relations in a lossless-join rnanner can be so deC0111pOsed into three or rnore relations. An example is a relation with attributes *supplier*, *part*, and *project*, denoted *SPJ*, with no FDs or MVDs. The JD [xI {SP, PJ, JS} holds.

Frorn the JD, the set of relation scheines *SP*, *PJ*, and *JS* is a IORsless-join decornposition of *SPJ*. Construct an instance of *HPJ* to illustrate that no two of these schernes suffice.

Exercise 19.15 Answer the following questions

1. Prove that the algorithrn shown in Figure 19.4 correctly cornputes the attribute closure of the input attribute set X.
2. Describe a linear-tirne (in the size of the set of FI)s, where the size of each FD is the nurnber of attributes involved) algoritlun for finding the attribute closure of a set of attributes with respect to a set of FDs. Prove that your algoritlun correctly COInputes the attribute closure of the input attribute set.

Exercise 19.16 Let us say that an 'Fl) $X \rightarrow Y$ is *simple* if Y is a single attribute.

1. Replace the FD $AB \rightarrow CD$ l.Jy the srnallest equivalent collection of sirnple FDs.
2. Prove that everyFD $X \rightarrow Y$ in a set of FDs F can be replaced by a set of sirnple F'Ds such that F^+ is equal to the closure of the new set of FDs.

Exercise 19.17 Prove that Arrnstrong's Axioms are sound and complete for FD inference. That is, show that repeated application of these axioms on a set F ofF'Ds produces exactly the dependencies in $P+$.

Exercise 19.18 Consider a relation R with attributes $ABCDE$. Let the following FDs be given: $A \rightarrow BC$, $Be \rightarrow E$, and $E \rightarrow DA$. Siluilarly, let S be a relation with attributes $ABCDE$ and let the follo\ving FDs be given: $A \rightarrow BC$, $B \rightarrow E$, and $E \rightarrow DA$. (Only the second dependency differs frolll those that hold over R.) You do not know whether or which other (join) dependencies hold.

1. Is R in BCNF?
2. Is R in 4NF?
3. Is R in 5NF?
4. Is Sin BeNF?
5. Is Sin 4NF?
6. Is Sin 5NF'?

Exercise 19.19 Let R be a relation schelna with a set F of FDs. Prove that the decOIll-position of R into HI and $R2$ is lossless-join if and only if $p+$ contains HI $\cap R_2 \rightarrow R_1$ or $R_1 \cap R2 \rightarrow R_2$.

Exercise 19.20 Consider a scheme R with FDs F that is decOlnposed into schelnes with attributes X and Y. Show that this is dependency-preserving if $F' \subseteq (F_X \cup py)+$.

Exercise 19.21 Prove that the optiInizatioll of the algorithrn for lossless··-join, dependency-preserving decornposition into 3NF relations (Section 19.6.2) is correct.

Exercise 19.22 Prove that the 3NF synthesis algoritlull produces a lossless-join decOlnposi-tion of the relation containing all the original attributes.

Exercise 19.23 Prove that an MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie \{XY, X(R - Y)\}$.

Exercise 19.24 Prove that, if R has only one key, it is in BCNF if and only if it is in 3NF.

Exercise 19.25 Prove that, if R is in 3NF and every key is shnple, then R is in HeNF.

Exercise 19.26 Prove these staternents:

1. If a relation scherne is in BCNF and at least one of its keys consists of a single attrilmte, it is also in 4NF.
2. If a relation scherne is in 3NF and each key has a single attribute, it is also in 5NF'.

Exercise 19.27 Give an algorithrn for testing whether a relation scheme is in BCNF. The :.llgorithrn should l)e polynorniaJ in the size of the set of given FDs. (The *size* is the surn over all FI)s of the nurnber of attributes that appear in theFJ).) Is there a polyuOlnial algorithrn for testing whether a relation scheme is in 3NF?

Exercise 19.28 Give an algorithm for testing whether a relation scherne is in BCNF. The algorithm should be polynomial in the size of the set of given FI)s. (The 'size' is the SUln over all FI)s of the nurnber of attributes that appear in theFD.) Is there a polynomial algorithrn for testing whether a relation scheme is in 3NF?

Exercise 19.29 1)rove that the algorithm for decomposing a relation scherna with a set of FI)s into a collection of BCNS relation schemas as describerl in Section 19.6.1 is correct (i.e., it produces a collection of BCNF relations, and is lossless-join) and terrninates.

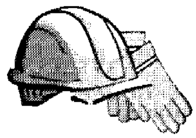
BIBLIOGRAPHIC NOTES

Textbook presentations of dependency theory and its use in database design include [3, 45, 501) 509, 747]. Good survey articles on the topic include [755, 415].

FDs were introduced in [187], along with the concept of 3NF, and aximlls for inferring FDs were presented in [38]. BCNF was introduced in [188]. The concept of a legal relation instance and dependency satisfaction are studied fonnaUy in [328]. FDs were generalized to scrnantic data Illodels in [768].

Finding a key is shown to be NP-COlnplete in [497]. Lossless-join decOlnpositions were studied in [28, 502, 627]. Dependency-preserving decorIlpositions were studied in [74]. [81] introduced rninirnal covers. DecOlnposition into 3NF is studied by 1.81, 98] and decOlnposition into BCNF is addressed in [742]. [412] shows that testing whether a relation is in 3NF is NP-cOlnplete. [253] introduced 4NF and discussed decoillposition into 4NF. Fagin introduced other nonnal forIlls in [254] (project-join nonnal fonn) and [255] (doHluin-key nonnal forIll). **In** contrast to the extensive study of vertical decOlnpositions, there has been relatively little formal investigation of horizontal decompositions. [209] investigates horizontal decoillpositiolls.

IvIVDs were discovered independently by Delobel [211], Fagin [253], and Zaniolo [789]. AxiOlllS for FDs and MVDs were presented in [73}. [593] shows that there is no axiornatization for JDs, although [662] provides an axioHlatization for a more general class of dependencies. The sufficient conditions for 4NF and 5NF in tenns of FDs that were discussed in Section 19.8 are from [205]. An approach to database design that uses dependency infornation to construct sarnple relation instances is described in [508, 509].



20

PHYSICAL DATABASE DESIGN AND TUNING

- What is physical database design?
- What is a query workload?
- How do we choose indexes? What tools are available?
- What is co-clustering and how is it used?
- What are the choices in tuning a database?
- How do we tune queries and view?
- What is the impact of concurrency on perforrnance?
- How can we reduce lock contention and hotspots?
- What are popular database benchnlarks and how are they used?
- Key concepts: Physical database design, database tuning, workload, co-clustering, index tuning, tuning wizard, index configuration, hot spot, lock contention, database benchmark, transactions per second

> Advice to a client who cornplained al)out rain leaking through the roof onto the dining table: "Move the table."
>
> —Architect Frank Lloyd Wright

The perfonnance of a DBMS on cornrnonly asked queries and typical update operations is the ultirnate Ineasure of a database desigIl. A I}BA can irnprove perforrnance by identifying perforrnance bottlenecks and adjusting sorne DBIVIS pararneters (e.g., the size of the buffer pool or the frequency of checkpointing) or adding hardware to elirninate such bottlenecks. rIhe first step in achieving good perforlnancc, however, is to Inake good database design choices, which is the focus of this chapter.

After we design the *conceptual* and *external* schernas, that is, create a collection of relations and views along 'with a set of integrity constraints, we .Il1ust address pe1'forlna11c8 goals through physical database design, in which we design the *physical* sche111a. As user requirernents evolve, it is usually necessary to tune, or adjust, all aspects of a database design for good performance.

This chapter is organized as follows. We give an overview of physical database design and tuning in Section 20.1. The 1nost irnportant physical design decisions concern the choice of indexes. We present guidelines for deciding which indexes to create in Section 20.2. These guidelines are illustrated through several exalnples and developed further in Sections 20.3. In Section 20.4, we look closely at the irnportant issue of clustering; we discuss how to choose clustered indexes and whether to store tuples fron1 different relations near each other (an option supported by sorne DBMSs). In Section 20.5, we emphasize how well-chosen indexes can enable some queries to be answered without ever looking at the actual data records. Section 20.6 discusses tools that can help the DBA to autornatically select indexes.

In Section 20.7, we survey the lnain issues of database tuning. In addition to tuning indexes, we 111ay have to tune the conceptual schema as well as frequently used query and view definitions. We discuss how to refine the conceptual schelna in Section 20.8 and how to refine queries and view definitions in Section 20.9. We briefly discuss the perforrnance irnpact of concurrent access in Section 20.10. We illustrate tuning on our Internet shop exarnple in Section 20.11. We conclude the chapter with a short discussion of DBMS benchrnarks in Section 20.12; benchrnarks help evaluate the perfOI'lnanCe of alternative DBl\1S products.

20.1 INTRODUCTION TO PHYSICAL DATABASE DESIGN

Like all other aspects of database design, physical design rnust be guided by the nature of the data a,nd its intended use. In particular, it is irnportant tonnderstand the typical **workload** that the database lI1Ust support; the workload consists of a rnix of queries and updates. Users also have certain requirements about how fast certain queries or updates 11111st run or how rnan.y tran.sactions rnust be processed per second. The \vorkload description and users' performance reqllirernents are the basis on \vhich a nurnber of decisions have to be rnade during pl1ysical database design.

Identifying Perfornlance Bottlenecks: All commercial systems provide a suite of tools for rnonitoring a wide range of systeIll parameters. These tools, used properly, can help identify perfofIna.nce bOttlenecks and suggest aspects of the database design and application code that need to be tuned for perforlnance. For example, we can ask the DBMS to rnonitor the execution of the database for a certain period of tinle and report on the nurnber of clustered scans, open cursors, lock requests, checkpoints, buffer scans, average wait titne for locks, and many such statistics that give detailed insight into a *snapshot* of the live system. In Oracle, a report containing this inforlnation can be generated by running a script called UTLBSTAT. SQL to initiate monitoring and a script UTLBSTAT. SQL to terminate rnonitoring. The system catalog contains details about the sizes of tables, the distribution of values in index keys, and the like. The plan generated by the DBMS for a given query can be viewed in a graphical display that shows the estimated cost for each plan operator. While the details are specific to each vendor, all InajaI' DBMS products on the market today provide a suite of such tools.

To create a, good physical database design and tune the systenl for performance in response to evolving user requirelnents, a designer HUlst understand the workings of a DBMS, especially the indexing and query processing techniques supported by the DBMS. If the database is expected to be accessed concurrently by rnany users, or is a *distributed database,* the task beeornes lnore cornplicated and other features of a, DBl\1S CaIne into play. We discuss the ilnpact of concurrency on database design in Section 20.10 and distributed databases in Chapter 22.

20.1.1 Database Workloads

The key to good physical design is arriving at an accurate description of the expectedworkloa.d. A workload description includes the follCJ\ving:

1. A list of queries (with their frequency, as a ratio of all queries / updates).
2. A list of updates and their frequencies.
3. Performance goals for each type of query and update.

For each quer.y in the workload, we HUlst identify

- Which relations are accessed.
- \Vhich attributes are retained (in the SELECT clause).

- \Vhich attributes have selection Of join conditions expressed on thern (in the WHERE clause) and how selective these conditions are likely to be.

SiInilarly, for each update in the \vorkloacl, we Blust identify

- Which attributes have selection or join conditions expressed on therll (in the WHERE clause) and how selective these conditions are likely to be.
- The type of update (INSERT, DELETE, or UPDATE) and the updated relation.
- For UPDATE cOHnuands, the fields that are rnodified by the update.

R.elllelubér that queries and updates typically have parameters, for exarnple, a debit or credit operation involves a particular account nUlnber. rrhe values of these paralneters deterlnine selectivity of selection and join conditions.

Updates have a query cornponent that is used to find the target tuples. This cOlllponent can benefit froIn a good physical design and the presence of indexes. On the other hand, updates typically require additional work to ITlaintain indexes on the attributes that they 111odify. Thus, while queries can only benefit froill the presence of an index, an index rnay either speed up or slow down a given update. Designers should keep this trade-off in rnind when creating indexes.

20.1.2 Physical Design and Tuning Decisions

Irnportant decisions rnade during physical database design and database tuning include the followving:

1. Choice of indexes to create:
   - Which relations to index and which field or cornbination of fields to choose as index search keys.
   - For each index, should it be clustered or ul1clustered?
2. *Tuning the conceptual schema:*
   - *Alternative normalized schemas:* We usually have rnore than one way to decompose a schclua into a desired IlOl'Inal fOITn (BCNF or 3NF). A choice can be rnade on the basis of perforrnance criteria.
   - *Denormalization:* We might want to reconsider scherna decolnpositions ca.rried OIlt for norrnalization. during the conceptual schern.a design process to irnprove the perforrnance of queries that involve attributes fr0111 several previously decornposed relations.

- *Vertical partitioning:* Under certain circurnstances we rnight 'want to further decornpose relations to ilnprove the perfornlance of queries that involve only a few attributes.
- *Views:* We luight 'want to add sorne views to nlask the changes in the conceptual scherna fr0111 users.

3. *Query and transaction tuning:* Frequently executed queries and transactions ulight be rewritten to run faster.

In parallel or distributed databases, which we discuss in Chapter 22, there are additional choices to consider, such as whether to partition a relation across different sites or whether to store copies of a relation at multiple sites.

20.1.3 Need for Database Thning

Accurate, detailed workload infonnation 111Cly be hard to corne by while doing the initial design of the systen1. Consequently, tuning a database after it has been designed and deployed is ilnportant---we HlllSt refine the initial design in the light of actual usage patterns to obtain the best possible perfonnance.

The distinction bet\veen database design and database tuning is soruewhat arbitrary. We could consider the design process to be over once an initial conceptual sche1na is designed and a set of indexing and clustering decisions is nlade. Any subsequent changes to the conceptual scherna or the indexes, say, would then be regarded as tuning. Alternatively, we could consider sorne refinernent of the conceptual scheula (and physical design decisions afl'ected by this refinernent) to be part of the physical design process.

Where we draw the line between design and tuning is not very irnpoItant, and we sirnply discuss the issues of index selection and database tuning without regard to when the tuning is carrier} out.

20.2 GUIDELINES FOR INDEX SELECTION

In considering which indexes to create, we begin with the list of queries (including queries tha,t a.ppear as part of update operations). ()bviously, only relations accessed by some query need to be considered as candidates for indexing, and the choice of attributes to index is guided by the conditions that appear in the WHERE clauses of the queries in the \vorkload. The presence of suitable indexes can significantly irnprove the evaluation plan for (1, query, as we saw in Chapters 8 and 12.

One approach to index selection is to consider the Ulost irnportant queries in turn, and, for each, deterrnine \vhich plan the optimizer would choose given the indexes currently on our list of (to be created) indexes. Then\ve consider whether we can arrive at a substantially better plan by adding more indexes; if so, these additional indexes are candidates for inclusion in our list of indexes. In general, range retrievals benefit froIn a B+ tree index, and exact-IHatch retrievals benefit frorn a hash index. Clustering benefits range queries, and it benefits exact-rnatch queries if several data entries contain the saIne key value.

Before adding an index to the list, however, we lnust consider the impact of having this index on the upda,tes in our workload. As we noted earlier, although an index can speed up the query cornponent of an update, all indexes on an updated attribute---{)n *any* attribute, in the case of inserts and deleteslnust be updated whenever the value of the attribute is changed. Therefore, we must sOlnetirnes consider the trade-off of slo\ving sorne update operations in the workload in order to speed up some queries.

Clearly, choosing a good set of indexes for a given workload requires an understanding of the available indexing techniques, and of the workings of the query optiruizer. The following guidelines for index selection sunnnarize our discussion:

Whether to Index (Guideline 1): The obviollS points are often the lnost important. Do not build an index unless sorne query including the query cOlnponents of updates benefits frolu it. Whenever possible, choose indexes that speed up rIlore than one query.

Choice of Search Key (Guideline 2): Attributes rnentioned ln a, WHERE clause are ca,ndidates for indexing.

- An exact-match selection condition suggests that we consider an irldex on the selected attributes, ideally, a hash index.

- j\ range selection condition suggests that we consider a 13+- tree (Of ISAM) index on the selected attrilnltes. j\ B+ tree index is usually preferable to an ISAM index. A.n ISAM irlclex rnay be worth considering if the relation is infrequently updated, but we assume that a B+ tree index is always chosen over an ISAM index, for sirnplicity.

Multi-Attribute Search :Keys (Guideline 3): Indexes with multiple-attribute search keys slH)uld be considered in the follc)\ving two situ<ltion.s:

- j\ WHERE clause includes conditinns 011 Inore t.han one attribute of a relation.

- They enable index-only evaluation strategies (i.e., accessing the relation can be avoided) for important queries. (This situation Gould lead to attributes being in the search key even if they do not appear in WHERE clauses.)

When creating indexes on search keys with rnultiple attributes, if range queries are expected, be careful to order the attributes in the search key to match the quenes.

Whether to Cluster (Guideline 4): At lllost one index on a given relation can be clustered, and clustering affects perfonnance greatly; so the choice of clustered index is iInportant.

- As a rule of tlnunb, range queries are likely to benefit the 1110St froIll clustering. If several ra,nge queries are posed on a relation, involving different sets of attributes, consider the selectivity of the queries and their relative frequency in the workload when deciding which index should be clustered.
- If an index enables an index-only evaluation strategy for the query it is intended to speed up, the index need not be clustered. (Clustering Inatters only when the index is used to retrieve tuples fr(nll the underlying relation.)

Hash versus Tree Index (Guideline 5): A B+ tree index is usually preferable because it supports range queries as well as equality queries. A hash index is better in the following situations:

- The index is intended to support index nested loops join; the indexed relation is the inner relation, and the search key includes the join colurllns. In this case, the slight ilnproveIllent of a hash index over a B+ tree for equality selections is rnagnified, because an equality selection is generated for each tuple in the outer relation..
- rrllcre is a very important equality query, and no range queries, involving the sea.rch key attributes.

Balancing the Cost of Index Maintenance (Guideline 6): After drawing up a 'wishlist' of indexes to create, consider the irnpact of each index on the updates in the workload.

- If maintaining an index slows down frequent update operations, consider dropping the index.
- Keep ill mind, however, that adding an index Illay 'well speed up a given update operation. For exanlplc, an index on employee IDs could speed up the operation of increasing the salary of a given ernployee (specified by ID).

20.3 BASIC EXAMPLES OF INDEX SELECTION

The follawing examples illustrate how to choose indexes during database design, continuing the discussion froln Chapter 8, where we focused on index selection for single-table queries. The schernas used in the exarnples are not described in detail; in general, they contain the attributes nalned in the queries. Additional inforlnation is presented when necessary.

Let us begin with a silnple query:

```
SELECT  E.enaIne, D.rugI'
FROM    Enlployees E, Departments D
WHERE   D.dname='Toy' AND E.dno=D.dno
```

The relations rnentioned in the query are Enlployees and Departnlents, and both conditions in the WHERE clause involve equalities. Our guidelines suggest that we should build hash indexes on the attributes involved. It seeIns clear that we should build a hash index on the *dnaTne* attribute of Departments. But consider the equality *E. dno=D. dno*. Should we build an index (hash, of course) on the *dno* attribute of Departrnents or Ernployees (or both)? Intuitively, we want to retrieve Departments tuples using the index on *dnarne* because few tuples are likely to satisfy the equality selection *.D. dnaTne= 'Toy'*.[1] For each qualifying Departrnents tuple, we then find lnatching EInployees tuples by using an index on the *dno* attribute of Ernployees. So, we should build an index on the *dno* field of En1ployees. (Note that nothing is gained by building an additional index on the *dno* field of Departrnents because Departnlents tuples are retrieved using the *dna:rne* index.)

Our choice of indexes was guided by the query evaluation plan we wanted to utilize. This consideration of a, potential evaluation plan is connnon while rnaking physical design decisions. U·nderstanding query optirnization is very useful for physical design. We show the desired plan for this query in Figure 20.1.

As a variant of this query, suppose that the WHERE clause is rnodified to be WHERE *J). dnarne= 'Toy'* AND *E'.dno=D. dno* AND *E.age=25*. Let us consider alternative evaluation plans. ()ne good plan is to retrieve Departrnents tuples that satisfy the selection on *dnarne* and retrieve rnatching Ernployees tuples by using an index on the *dno* field; the selection on *age* is then applied on-the-fly. However, unlike the previous variant of this query, we do not really need to have an index on the *dna* field of Ernployees if we have an index. on *age*. In this

[1]This is only a heuristic. If *dname* is not the key, and we have no statistics to verify this cla.inl. it is possible that several tuples satisfy this condition.

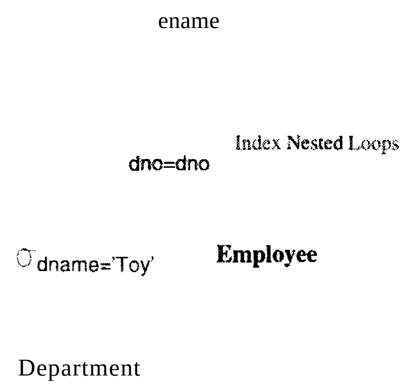


Figure 20.1 A Desirable Query Evaluation Plan

case we can retrieve Departrnents tuples that satisfy the selection on *dnarrte* (by using the index on *dname,* as before), retrieve Ernployees tuples that satisfy the selection on *age* by using the index on *age,* and join these sets of tuples. Since the sets of tuples we join are srnall, they fit in 111emory and the join Inethod is unirnportant. This plan is likely to be sornewhat poorer than using an index on *dno,* but it is a reasonable alternative. rrherefore, if we have an index on *age* already (prolnpted by sorne other query in the workload), this variant of the sarnple query does not justify creating an index on the *dno* field of Ernployees.

Our next query involves a range selection:

```
SELECT  E.enan1e, I),dnarne
FROM    Elnployees E, Departrnents D
WHERE   B.sal BETWEEN 10000 AND 20000
        AND E.hobby='Starups' AND E.dno=D.dno
```

This query illustrates the use of the BETWEEN operator for expreSSIng range selections. It is equivalent to the condition:

$$10000 \leq \text{E.sal AND E.sal} \leq 20000$$

The use of BETWEEN to express rarlge conditions is reconunended; it lnakes it easier for both the user and the optilnizer to recognize both parts of the range selection.

Returning to the exarnple query, both (nonjoin) selections are on the Ernployees relation. Therefore, it is clear that a plan in which Eluployees is the outer relation and I)epartrnents is the inner relation is the best, as in the previous query, and we should build a hash index on the *dno* attribute of Departlnents. But which index should we build on Ernployees? A B+ tree index on. the *sal* attribute would help with the range selection, especially if it is clustered. A hash index on the *hobby* attribute would help ·with the equality selection. If one of these indexes is available, we could retrieve Ernployees tuples using this index, retrieve rnatching Departruents tuples using the index on *dno*, aJld apply all rernaining selections and projections on-the-fly. If both indexes are available, the optirnizer would choose the mo1'e selective index for the given query; that is, it \vollld consider which selection (the range condition on *salary* or the equality on *hobby)* has fe\ver qualifying tuples. In general, which index is rnore selective depends on the data. If there are very few people with salaries in the given range and rnany people collect starnps, the B-t- tree index is best. Otherwise, the hash index on *hobby* is best.

If the query constants are known (as in our exarnple), the selectivities can be estiInated if statistics on the data are available. Otherwise, as a rule of thurnb, an equality selection is likely to be rnore selective, and a reasonable decision would be to create a hash index on *hobby.* Sornethnes, the query constants are not known—we rnight obtain a query by expanding a query on a view at rUIl-tirrle, or we rnight have a query in Dynalnic SQL, which allows constants to be specified as *wild-card variables* (e.g., *%X)* and instantiated at run-tinle (see Sections 6.1.3 and 6.2). In this case, if the query is very important, we lllight choose to create a B+ tree index on *sal* and a hash index on *hobby* and leave the choice to be rnade by the optirnizer at run-tirrle.

20.4 CLUSrrERING AND INDEXING

Clustered indexes can be especially iInportant while accessing the inner relation in an index nested loops join. To understand the relationship between clustered hldexes and joins, let us revisit our first exarnple:

```
SELECT E.enanle, D.rngr
FROM   Employees E, I)epartrnentsD
WHERE  D.dname='Toy' AND E.dno=D.dno
```

We concluded that a good evaluation plan is to use an index on *dname* to retrieve Departments tuples satisfying the condition on *dnarne* and to find. rnatching Ernployees tuples using an index on *dna*. Should these indexes be clustered? G·iven our asslunption that the number of tuples satisfying 1).*dname='Toy'* is likely to be small, we should build an unclustered index on *dname.* ()n the other hand, Employees is the inner relation in an index nested loops join and *dna* is not a candidate key. This situation is a strong argument that the index on the *dno* field of Ernployees 8ho111(1 be clustered. In fact, because the join consists of repeatedly posing equality selections on the dnofield of the inner relation, this type of quer,Y is a stronger justification for rnaking the index on *dno* clustered than a sirnple selection query such as the previous selection on *hobby.* (Of courso, factors such as selectivities and frequency of queries have to be taken into account as well.)

'rhe following oxaluplc, very sirnilar to the previous one, illustrates how clustered indexes can be used for sort-rnerge joins:

```
SELECT  E.enarne,D.rngr
FROM    Ernployees E, DepartlTlents D
WHERE   E.hobby='Starnps' AND E.dno=D.dno
```

This query differs froIll the previous query in that the condition *E.hobby='Starnps'* replaces *D.dname='Toy'.* Based on the assumption that there are few ernployees in the Toy departrnent, we chose indexes that would facilitate an indexed nested loops join with DepartlTlents as the outer relation. Now, let us suppose that rllany ernployees collect stamps. In this case, a block nested loops or sort-rnerge join Blight be rnore efficient. A sort-rnerge join can take advantage of a clustered B+ tree index on the *dno* attribute in Departrnents to retrieve tuples and thereby avoid sorting Departrnents. Note that an unelustered index is not useful----since all tuples are retrieved, performing one I/O per tuple is likely to be prohibitively expensive. If there is no index on the *dno* field of Ernployees, we could retrieve Ernployees tuples (possibly using an index on *hobby,* especially if the index is clustered), apply the selection *E.hobby='Starnps'* oll-the-fly, and sort the qualifying tuples on *dno.*

As our discllssion has indicated, when we retrieve tuples using an index, the irnpact of clustering depends on the rnunber of retrieved tuples, that is, the nuruber of tuples that satisfy the selection conditions that rnatch the index. An unclustered index is just as good as (1, clustered index for a selection that retrieves a single tuple (e.g., an equality selection on a candidate key). As the llurnber of retrieved tuples increases, the unclustered index quickly becoHlcs rnore expensive than e'ven a sequential scan of the entire relation. Although the sequential scan retrieves all tuples, each page is retrieved exactly once, whereas a page rIlay be retrieved as often as the rnunber of tuples it contains if an unclustered index is usee!' If blocked l/C) is perforrned (as is COl1nnon), the relative advantage of sequential scan versus an llnclustered index increases further. (Blocked T/C) also speeds up access using a clustered index, of course.)

We illustrate the relationship between the number of retrieved tuples, viewed as a percentage of the total nurnber of tuples in the relation, and the cost of various access rnethods in .Figure 20.2. We assume that the query is a selection on a single relation, for sirnplicity. (Note that this figure reflects the C()st of writing out the result; otherwise, the line for seqnential scan weHlld be flat.)

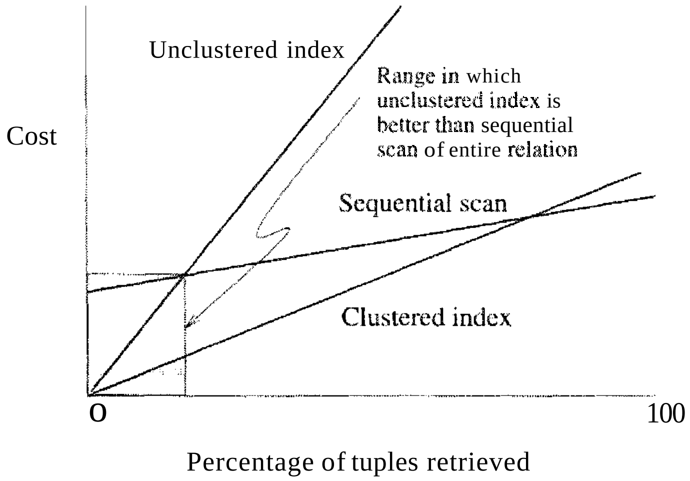


Figure 20.2 The Impact of Clustering

20.4.1 Co-clustering Two Relations

In our description of a typical database systern architecture in Chapter g, we explained how a relation is stored as a file of records. Although a file usually contains only the records of SOIn8 one relation, SCHue systeIlls allow records frorn Inore than one relation to be stored in a single file. rrhe database user can request that the records froIll two relations be interleaved physically in this 111anne1'. This data layout is sornetiInes referred to as co-clustering the two relations. We now discuss when co-clustering can be beneficial.

As an exarnple, consider two relations with the following schernas:

Parts(<u>pid: integer,</u> *pnarne:* string, *cost:* integer, *8upplierid:* integer)
<u>Assembly(*partid:*</u> integer, <u>*componentid:*</u> <u>integer,</u> *quantity:* integer)

In this scherna the *componentid* field of Assernbly is intended to be the *pid* of sorne part that is used as a cornponent in assernbling the part with *pid* equal to *partid.* Therefore, the Assernbly table represents a 1:N relationship between parts and their subparts; a part can have rnany s11bparts, but each part is the subpart of at rnost one part. In, the Parts table, *pid* is the key. For cOlnposite parts (those assernbled frorIl other parts, as indicated by the contents of Assclnb1y), the *cost* field is taken to be the cost of assembling the part frorn its subparts.

Suppose that a frequent query is to find the (inllnediate) subparts of all parts supplied by a given supplier:

```
SELECT P.piel, .A.componentid
FROM   Parts J'), Assembly A
```

```
WHERE   P.piel = A.partid AND P.supplierid = 'Acme'
```

A good evaluation plan is to apply the selection condition on Parts and then retrieve rnatching Asselnbly tuples throngh an index on the *partid* field. Ideally, the index on *partid* should be clustered. rrhis plan is reasonably good. :However, if such selections are COHU110n and we want to optirnize thorn further, we can *co-cluster* the two tables. In this approach, we store records of the two tables together, \vith each Parts record *P* follc)\ved by all the Assernbly records *A* such that *P.pid* = *A.partid*. This approach improves on storing the two relations separately and having a, clustered index on *paTtid* because it does not need an index lookup to find the A.ssernbly records that rnatch a given Parts record. Thus, for each selection query, we save a few (typically two or three) index page l/Os.

If we are interested in finding the imrnediate subparts of *all* parts (i.e., the preceding query with no selection on *supplierid)*, creating a clustered index on *partid* and doing an index nested loops join with Assembly as the inner relation offers good perfonnance. An even better strategy is to create a clustered index on the *paTtid* field of Assernbly and the *pid* field of Parts, then do a sort-rnerge join, using the indexes to retrieve tuples in sorted order. This strategy is comparable to doing the join using a co-clustered organization, which involves just one scan of the set of tuples (of Parts and Asselnbly, which are stored together in interleaved fashion).

The real benefit of co-clustering is illustrated by the following query:

```
SELECT  P.pid,A.componentid
FROM    Parts P, Assernbly A
WHERE   P.pid = A.partid AND P.cost=10
```

Suppose that rnany parts have *cost* = 10. This query essentially a,rnonnts to a collection of queries in which we are given a Parts record and want to find rnatching Assernbly records. If we have an index on the *cost* field of Parts, we can retrieve qualifying Parts tuples. For each such tuple, we have to use the index on Assernbly to locate records with the given *pid*. rrhe index access for A.ssernbly is avoided if we have a co-clustered organization. (()f courS8, we still require all index on the *cost* attribute of Parts tuples.)

Such an optirnization is especially irnportant if we want to traverse several levels of the part-subpart hierarchy. .For example, a COnll110Il query is to find the totaJ cost of a part, which requires us to repeatedly carry out joins of Pa,rts and Asscrnbly. Incidentally, if we do not know the nurnber of levels in the hierarchy ill advance, the nUlnber of joins varies and the query cannot be expressed in SQL. The query can be answered by ernbedcling an SQL statement for the join insicie an iterative host language prograrll. How to express the query is orthogonal to our lnain point here, which is that co-clustering is especially beneficial when the join in question is carried out very frequently (either because it arises repeatedly in an important query such as finding total cost, or because the join query itself is asked frequently).

To sUIlunarize co-clustering:

- It can speed up joins, in particular key·foreign key joins corresponding to l:N relationships.
- A sequential scan of either relation becornes slower. (In our exalnple, since several Assenlbly tuples are stored in between consecutive Parts tuples, a scan of all Parts tuples becornes slower than if Parts tuples were stored separately. SiInilarly, a sequential scan of all Assernbly tuples is also slower.)
- All inserts, deletes, and updates that alter record lengths becorne slower, thanks to the overheads involved in ruaintaining the clustering. (We do not discuss the irnplernentation issues involved in co-clustering.)

20.5 INDEXES THAT ENABLE INDEX-ONLY PLANS

This section considers a nU111ber of queries for which we can find efficient plans that avoid retrieving tuples froln one of the referenced relations; instead, these plans scan an associated index (which is likely to be lnuch srnaller). An index that is used (only) for index-only scans does *not* have to be clustered because tuples fronl the indexed relation are not retrieved.

This query retrieves the lnanagers of depal'truents with at least one ernployee:

```
SELECTD.rugr
FROM    Departrnents I), Employees E
WHERE   I).dno=E.dno
```

()bserve that no attributes of Ernployees are retained. If we have an index on the *dno* field of Employees, tl1e optirnization of doirlg an index nested loops join using an index-onl:y searl for the inner relation is applicable. Given this variant of the query, the correct decision is to build an uncIustered index on the *dna* field of Employees, rather thaIl a clustered index.

rrhe next query takes this idea a step further:

```
SELECT  ]).rngr, E.eid
FROM    ])epartrnents I), Employees E
WHERE   D.dno=E.dno
```

If we have an index on the *dno* field of Employees, we can use it to retrieve Employees tuples during the join (\vith Departrnents as the outer relation), but unless the index is clustered, this approach is not be efficient. ()n the other hand, suppose that we have a B+- tree index on *(dna, e'id)*.*Now* all the inforrnation we need about an Employees tuple is contained in the data entry for this tuple in the index. We can use the index to find the first data entry \vith a given *elno;* all data entries 'with the same *dno* are stored together in the index. (Note that a hash index on the cOlnposite key *(dna, eid)* cannot be used to locate an entry with just a given *dno!)* We can therefore evaluate this query using an index nested loops join with Departlnents as the outer relation and an index-only scan of the inner relation.

20.6 TOOLS TO ASSIST IN INDEX SELEC"fION

The rUllnber of possible indexes to consider building is potentially very large: For each relation, we can potentially consider all possible subsets of attributes as an index key; we have to decide on the ordering of the attributes in the index; and we also have to decide which indexes should be clustered and which unclustered. Many large applications---for exalnple enterprise resource planning systems---create tens of thousands of different relations, and rnanual tuning of such a large schelna is a daunting endeavor.

The difficulty and irnportance of the index selection task rnotivated the development of tools that help database adrninistrators select appropriate indexes for a given workload. The first generation of such **index tuning wizards**, or **index advisors**, were separate tools outside the database engine; they suggested indexes to build, given a workload of SQL queries. rfhe rnain drawback of these systerns was that they had to replicate the database query optirnizer's cost rnodel in the tuning tool to rnake sure that the optimizer would choose the sanlC query evaluation plans as the design tool. Since query optirnizers cha,nge froIn release to release of a conunercial database systern, considerable effort was needed to keep the tuning tool and the database optirnizer synchronized. The rnost recent generation of tuning tools are integrated \vith the database engine and use the database query optiluizer to estimate the cost of a workload given a set of indexes, avoiding duplication of the query optirnizer's cost rnodel into an external tool.

20.6.1 Automatic Index Selection

\{Ve call a set of indexes for a given database scherna. an **index configuration**. We assume that a, query workload is a set of queries over a database scherna where each query has a frequency of occurrence assigned to it. Given a database schelna and a, workload, the **cost of an index configuration** is the expected cost of running the queries in the workload given the index configuration taking the different frequencies of queries in the workload into account. Given a database schema and a query workload, we can now define the problel11 of automatic index selection as finding an index configuration with nlinirnal cost. A.s in query optinlization, in practice our goaJ is to find a *good* index configuration rather than the true optirnal configuration.

Why is automatic index selection a hard problern? Let us calculate the nUlnber of different indexes with c attributes, assurning that the table has n attributes. For the first attribute in the index, there are n choices, for the second attribute $n-1$, and thus for a c attribute index, there are overall $n \cdot (n-1) \ldots (n-c+1) = \frac{n!}{(n-c)!}$ different indexes possible. The total nurnber of different indexes with up to c attributes is

$$\sum_{i=1}^{c} \frac{n!}{(n-i)!}$$

For a table with 10 attributes, there are 10 different one-attribute indexes, 90 different two-attribute indexes, and 30240 different five-attribute indexes. For a cornplex workload involving hundreds of tables, the nurnber of possible index configurations is clearly very large.

The efficiency of autornatic index selection tools can be separated into two components: (1) the nurnber of candidate index configurations considered, and (2) the nurnber of optimizer calls necessary to evaluate the cost for a configuration. Note that reducing the search space of candidate indexes is analogous to restricting the search space of the query optiInizer to left-deep plans. In lnany cases, the optirnal plan is not left-deep, but alllong all left-deep plans there is usually a plan whose cost is close to the optirnal plan.

We can easily reduce the time taken for autornatic index selection by reducing the nUlnber of candidate index configurations, but the srnaller the space of index. configurations considered, the farther away the final index configuration is [1'0111 the optirnal index configllration. rrherefore, different index tuning \vizards prune the search space differently, for exarnple, by considering only one- or two-attribute indexes.

20.6.2 How Do Index Thning Wizards Work?

All index tuning \vizards search a set of candidate indexes for an index configuration with lowest cost. Tools differ in the space of candidate index configurations they consider and how they search this space. We describe one representative algoritlun; existing tools implement 'variants of this algorithrn, hut their irnplernentations have the sanIe basic structure.

The DB2 Index Advisor. The DB2 Index Advisor is a tool for automatic index recommendation given a workload. The workload is stored in the database systell1 in a table called ADVISE_WORKLOAD. It is populated either (1) by SQL statcrnents 1'1'0111 the DB2 dynanlic SQL statelnent cache, a cache for recently executed SQL statenlents, (2) with SQL staternents frol11 packages--groups of statically cornpiled SQL statenlents, or (3) with SQL statelnents frolIl an online monitor called the Query Patroller. The DB2 Advisor allows the user to specify the lnaximuill arnount of disk space for new indexes and a rnaxirrnul1 time for the cornputation of the recomrnended index configuration.

The DB2 Index Advisor consists of a prograrrl that intelligently searches a subset of index configurations. Given a candidate configuration, it calles the query optirnizer for each query in the ADVISE_WORKLOAD table first in the RECOMMEND_INDEXES rnode, where the opthnizer recommends a set of indexes and stores thern in the ADVISE_INDEXES table. In the EVALUATE_INDEXES mode, the optimizer evaluates the benefit of the index configuration for each query in the ADVISE-WORKLOAD table. The output of the index tuning step is are SQL DDL statenlents whose execution creates the recomrnended indexes.

The Microsoft SQL Server 2000 Index Tuning Wizard. Microsoft pioneered the irnplelll1entation of a tuning wizard integrated with the database query optiInizer. The Microsoft Tuning vVizard has three tuning rnodes tha.t perrnit the user to trade off running time of the analysis and nurnber of candidate index configurations exarnined: *fast*, *medium*, and *thorough*, with *fast* having the lo\vest running tirne aJld *thoTo'ugh* exalnining the largest nUlnber of configurations. rro further reduce the running time, the tool has a salnpling Inode in which the tuning wizard randoruly salllpics queries fronl the input workload to speed up analysis. Other parameters include the lnaxirnurn space allowed for the reeornmended indexes, the. maximum nurnber of attributes per index considered, and the tables on which Indexes can. be generated. The Microsoft Index Tuning WIzard also, perunts *table scaling*, where the user can specify an anticipated nurnber of records for the tables involved in the workload. This allows users to plan for future growth of the tables.

Before we describe the index tuning algoriUlIn, let us consider the problell1 of estiInating the cost of a configuration. Note that it is not feasible to actually create the set of indexes in a candidate configuration and then optirnize the query workload given the physical index configuration. Creation of even a single candidate configuration with several indexes lIlight take hours for large databases and put considerable load on the database systerIl itself. Since we want to exauline a large nUlnber of possible candidate configurations, this approach is not feasible.

Therefore index tuning algorithrIls usually *simulate* the effect of indexes in a candidate configuration (unless such indexes already exist). Such **what-if** indexes look to the query optilIlizer like any other index and are taken into account when calculating the cost of the workload for a given configuration, but the creation of what-if indexes does not incur the overhead of actual index creation. Commercial database systelIls that support index tuning wizards using the database query optirnizer have been extended with a module that permits the creation and deletion of what-if indexes with the necessary statistics about the indexes (that are used when estirnating the cost of a query plan).

We now describe a representative index tuning algorithm. The algorithm proceeds in two steps, *candidate index selection* and *configuration enumeration*. In the first step, we select a set of candidate indexes to consider during the second step as building blocks for index configurations. Let us discuss these two steps in Inore detail.

Candidate Index Selection

We saw in the previous section that it is iInpossible to consider every possible index, due to the huge nUluber of candidate indexes available for larger database schernas. ()ne heuristic to prune the large space of possible indexes is to first tune each query in the workload independently and then select the union of the indexes selected in this first step as input to the second step.

:For a query, let us introduce the notion of an indexable attribute, which is an attribute whose appearance in an index could change the cost of the query. An **indexable attribute** is an attribute on \vhich the WHERE-part of the query has a condition (e.g., an equality predicate) or the attribute appears in a GROUP BY or ORDER BY clause of the SQL query. An **admissible index** for a query is an index that contains only indexable attributes in the query.

How do we select candidate indexes for an individual query? ()ne approach is a basic enumeration of all indexes with up to k attributes. \Ve start \ivith all indexable attributes as single attribute candidate indexes, then add all combinations of two indexable attributes as candidate indexes, and repeat this procedure until a user-defined size threshold k. This procedure is obviously very expensive as we add overall $n + n \cdot (n - 1) + \ldots + n \cdot (n - 1) \ldots (n - k + 1)$ candidate indexes, but it guarantees that the best index with up to k attributes is aIl10ng the candidate indexes. The references at the end of this chapter coll-tain pointers to faster (but less exhaustive) heuristieal search algorithrns.

Enumerating Index Configurations

In the second phase, we use the candidate indexes to enUInerate index configurations. As in the first phase, we can exhaustively enurnerate all index configurations up to size k, this time cornbining candidate indexes. As in the previous phase, more sophisticated search strategies are possible that cut down the number of configurations considered while still generating a final configuration of high quality (i.e., low execution cost for the final workload).

20.7 OVERVIEW OF DATABASE TUNING

After the initial phase of database design, actual use of the database provides a valuable source of detailed information that can be used to refine the initial design. Many of the original assumptions about the expected workload can be replaced by observed usage patterns; in general, some of the initial workload specification is validated, and some of it turns out to be wrong. Initial guesses about the size of data can be replaced with actual statistics frorn the systern catalogs (although this inforrnation keeps changing as the systern evolves). Carefulrnonitoring of queries can reveal unexpected problerlls; for example, the optimizer lllay not be using SOIne indexes as intended to produce good plans.

Continued database tuning is irnportant to get the best possible perforrnance. In this section, we introduce three kinds of tuning: *tuning indexes, tuning the conceptual scherna,* and *tuning queries.* oUf discussion of index selection also applies to index tuning decisions. Conceptual schcrna and query tuning are discussed further in Sections 20.8 and 20.9.

20.7.1 Thning Indexes

The initial choice of indexes rnay be refined for one of several reasons. The sirnplest reason is that the observed \vorkload reveals that scnne queries and updates considered irnportant in the initial\vorkload specification are not very frequent. The observed workload rnay a1so identify sCHne new queries and updates that *are* inlportant.The initial choice of indexes has to be reviewed in light of this new infornation. Scnne of the original in,dexes rnay be dropped and new ones added. The reasoning involved is siInilar to that used in the initial design.

It Inay also be discovered that the optimizer in a given systenl is not finding some of the plans that it was expected to. For exaluple, consider the following query, which we discussed earlier:

```
SELECT  D.Ingr
FROM    Ernployees E, Departulents D
WHERE   D.dnamc='Toy' AND E.dno=D.dno
```

A good plan here would be to use an index on *dnarne* to retrieve Departnlents tuples with *dnarne= 'Toy'* and to use an index on the *dno* field of Employees as the inner relation, using an index-only scan. Anticipating that the optirnizer would find such a plan, we rnight have created an unclustered index on the *dno* field of Ernployees.

Now suppose queries of this fonn take an unexpectedly long time to execute. We can ask to see the plan produced by the optiInizer. (Most commercial systerIls provide a simple cOillrnand to do this.) If the plan indicates that an index-only scan is not being used, but that Employees tuples are being retrieved, we have to rethink our initial choice of index, given this revelation about our system's (unfortunate) lhnitations. An alternative to consider here would be to drop the unclustered index on the *dno* field of EUlployees and replace it with a clustered index.

SOUle other COlllllnon lirnitations of optiInizers are that they do not handle selections involving string expressions, arithrnetic, or *null* values effectively. We discuss these points further when we consider query tuning in Section 20.9.

In addition to re-exarnining our choice of indexes, it pays to periodically reorganize S(Hne indexes. For example, a static index, such as an ISAM index, Illay have developed long overflow chains. Dropping the index and rebuilding it—if feasible, given the interrupted access to the indexed relation—can substantially irnprove access tiTHes through this index. Even for a dynarnic structure such as a 13+ tree, if the implernentation does not rnerge pages on deletes, space occupancy can decrease considerably in SaIne situations. This in turn rnakes the size of the index (in pages) larger than necessary, and could increase the height and therefore the access tilne. Ilebuilding the index should be considered.Extensive updates to a clustered index rnight also lead to overflow pages being allocated, thereby decreasing the degree of clustering. Again, rehuilding the index Inay be vvorthwhile.

Finally, note that the query optinlizer relies on statistics rnaintained in the SystCIll catalogs. These statistics are updated only when a special utility progranl is run; be sure to run the utility frequently enough to keep the statistics reasonably current.

20.7.2 Thning the Conceptual Schema

In the course of database design, we rnay realize that our current choice of relation SChelllaS does not enable us rneet our perforrnance objectives for the given workload with any (feasible) set of physical design choices. If so, we 11lay have to redesign our conceptual scherna (and re-exarnine physical design decisions affected by the changes we rnake).

We rnay realize that a redesign is necessary during the initial design process or later, after the systern has been in use for a while. Once a database has been designed and populated with tuples, changing the conceptual scherna requires a significant effort in tenrlS of rnapping the contents of the relations affected. Nonetheless, it rnay be necessary to revise the conceptual scherna in light of experience with the systern. (Such changes to the schema of an operational systerll are sometirnes referred to as **schema evolution**.) We now consider the issues involved in conceptual scherna (re)design frorn the point of vie\v of perforrnance.

rrhe rnain point to understand is that *our choice of conceptual 8cherna should be guided by a cons'ideration of the queries and 'updates in our 'workload"* in addition to the issues of redundancy that rllotivate nonnalization (which we discussed in Chapter 19). Several options rnust be considered while tuning the conceptual scherna:

- We lIlay decide to settle for a 3NF design instead of a BCN'F design.
- If there are two ways to decornpose a given schelna into 3NF or BCNF, our choice should be guided by the workload.
- Sometimes we rnight decide to further decornpose a relation that is *already* rn BCNF.
- In other situations, we rnight *denormalize.* rrhat is, we rnight choose to replace a collection of relations obtained by a dec(nnposition frorn a larger relation with the original (larger) relation, even though it suffers frorn 80rne redundancy problerl1s. Alternatively, we rnight choose to add sorne fields to certain relations to speed up SCHne irnportant queries, even if this leads to a redundant storage of 80rne infonnation (anei, consequently, a scherna that is in neither 3NF nor BCNF).

- This discussion of nonnalization has concentrated on the technique of *de-composition*, 'which arnounts to vertical partitioning of a relation. Another technique to consider is *horizontal partitioning* of a relation, which \vould lead to having two relations with identical schernas. Note that we are not talking about physically partitioning the tuples of a single relation; rather, we want to create two distinct relations (possibly with different constraints and indexes on each).

Incidentally, when \ve redesign the conceptual scherna, especially if we are tun-ing an existing database sche1na, it is \vorth considering whether we should create views to rnask these changes fronl users for WhOIII the original schcrlla is]nore natural. We discuss the choices involved in tuning the conceptual scherna in Section 20.8.

20.7.3 Thning Queries and Views

If we notice that a query is running rnuch slower than we expected, we have to exarnine the query carefully to find the problern. SaIne rewriting of the query, perhaps in conjunction with SCHne index tuning, can often fix the problern. Sirn-ilar tuning rnay be called for if queries on SaIne view run slower than expected. We do not discuss view tuning separately; just think of queries on views as queries in their own right (after all, queries on views are expanded to account for the view definition before being optirnized) and consider how to tune thern.

When tuning a query, the first thing to verify is that the systern uses the plan you expect it to use. Perhaps the systelll is not finding the best plan for a variety of reasons. Sorne COIllrllon situations not handled efficiently by rnany optinlizers follow:

- A selection condition involving *null* values.

- Selection conditions involving aritlunetic or string expressions or concli-tions using the OR connective. For exarnple, if we have a conclitionE.*age* = 2*]).*age* in the WHERE clause, the optirnizer rnay correctly utilize an available index on *E.age* but fail to utilize an available index on 1).*age*. R,eplacing the condition by *E.age/2* = 1).*age* \vould reverse the situation.

- Inability to recognize a sophisticated plan such as an index-only scan for an aggregation query involving a GROUP BY clause. ()f course, virtually no optirnizer looks for plans outside the plan space described in Chapters 12 and 15, such as nonleft-deep join trees. So a good understanding of ·what an optirnizer typically does is irnportant. In addition, the rnore a:ware you are of a given systeur's strengths arrd lirnitations, the better off you arc.

If the optirnizer is not smart enough to find the best plan (using access Inethods and evaluation strategies supported by the DB.wIS), sOHIe systern.s allow users to guide the choice of a plan by providing hints to the opthnizer; for example, users rnight be able to force the use of a particular index or choose the join order and join rnethod. A user who wishes to guide optirnization in this Inanner should have a thorough understanding of both optirnizatioll and the capabilities of the given DBMS. We discuss query tuning further in Section 20.9.

20.8 CHOICES IN TUNING THE CONCEPTUAL SCHEMA

We now illustrate the choices involved in tuning the conceptual scheIua through several exarnples using the following schelnas:

Contracts(*cid:* integer, *s'Upplierid:* integer, *projectid:* integer,
deptid: integer, *partid:* integer, *qty:* integer, *value:* real)
Departments(*did:* integer, *budget:* real, *annualreport:* varchar)
Parts(*pid:* integer, *cost:* integer)
Projects(*jid:* integer, *rngr:* char(20))
Suppliers(*sid:* integer, *address:* char(50))

For brevity, we often use the cornrnon convention of denoting attributes by a single character and denoting relation schernas by a sequence of characters. Consider the scherna for the relation Contracts, whic.h we denote as CSJDPQV, with each letter denoting an attribute. The Ineaning of a tuple in this relation is that the contract with *cid* C is an agreernent that supplier S (with *sid* equal to *supplierid)* will supply Q iterns of part P (with *pid* equal to *partid)* to project J (with *jid* equal to *projectid*) associated with departrnent D (with *deptid* equal to *did),* and that the value V of this contract is equal to *value.*[2]

There are two known integrity constraints with respect to Contracts. A project purchases a given part using a single contract; thus, there cannnot be two distinct contracts in which the saIne project buys the saIne part. This constraint is represented using th.e FI) *.II)* $\rightarrow C$. Also, a departrnent purchases at rnost one part frolll any given supplier. This constraint is represented llsing the FD $SD \rightarrow P$. In addition, of course, the contract ID C is a key. The rneaning of the other relations should be obvious, and we do not describe thcrn further because we focus on the Contra.cts rela.tion.

2If this schema seems cornplicated, note that real-life situations often call for considerably more cornplex schemas!

20.8.1 Settling for a Weaker Normal Form

Consider the Contracts relation. Should we deconlpose it into sHlaller relations? Let us see what norrnal fann it is in. The candidate keys for this relation are C and JP. (C is given to be a key, and tIP functionally deterrnines C.) The only nonkey dependency is $3D \rightarrow P$, and P is a *prime* attribute because it is part of candidate key JP. rrhus, the relation is not in BCNF—because there is a nonkey dependency—but it is in 3NF.

By using the dependency $SD \rightarrow P$ to guide the decornposition, we get the two schemas SDP and CSJDQV. This decornposition is lossless, but it is not dependency-preserving. However, by adding the relation schelne CJP, we obtain a lossiess-join, dependency-preserving decoruposition into BCNF. Using the guideline that such a decorllposition into BCNF is good, we might decide to replace Contracts by three relations with schernas CJP, SDP, and CSJDQV.

However, suppose that the following query is very frequently asked: Find the nurnber of copies Q of part P ordered in contract C. 'rhis query requires a join of the decornposed relations CJP and CSJDQV (or SDP and CSJDQV), whereas it can be answered directly using the relation Contracts. The added cost for this query could persuade us to settle for a 3NF design and not decompose Contracts further.

20.8.2 Denormalization

The reasons rTIotivating us to settle for a weaker norrnal forIn may lead us to take an even rnore extrerne step: deliberately introduce SOlllC redundancy. As an exarnple, consider the Contracts relation, 'which is in 3NF. Now, suppose that a frequent query is to check that the value of a contract is less than the budget of the contracting departruent. We lllight decide to add a budget field B to Contracts. Since *did* is a key for Departrnents, we now have the dependency $D \rightarrow B$ in Contracts, \vhich InCt1IlS Contracts is not in 3NF any r11orc. Nonetheless, we rnight choose to stay with this design if the rnotivating query is sufficiently irnportant. Such a decision is clearly subjective and CaInes at the cost of significant redundancy.

20.8.3 Choice of Decomposition

Consider the Contracts relation again. Several choices are possible for dealing with the redundancy in this relation:

- We can leave Contracts as it is and accept the redundancy associated \\"ith its being in :3N:F rather than .BCNF.

- We Inight decide that we want to avoid the anolIlalies resulting froIn this redundancy by deeornposing Contracts into BCNF using one of the following ruethods:
  - We have a lossless-join decornposition into PartInfo with attributes SDP and Contractlnfo \vith attributes CSJDQV. As noted previously, this decornposition is not dependency-preserving, and to rnake it so would require us to add a third relation CJP, \vhose sale purpose is to allow us to cheek the dependency $JP \rightarrow C$.
  - We could choose to replace Contracts by just Partlnfo and Contract-Info even though this decornposition is not dependency-preserving.

R,eplacing Contracts by just Partlnfo and Contractlnfo does not prevent us frorll enforcing the constraint $JP \rightarrow C$; it only makes this n10re expensive. We could create an assertion in SQL-92 to check this constraint:

```
CREATE ASSERTION checkDep
CHECK    ( NOT EXISTS
         (SELECT  *
         FROM     Partlnfo PI, Contractlnfo Cl
         WHERE    PI.supplierid=CI.supplierid
                  AND PI.deptid==CI.deptid
         GROUP BY C1.projectid, PI.partid
         HAVING   COUNT (cid) > 1 ) )
```

This assertion is expensive to evaluate because it involves a join followed by a sort (to do the grouping). In cornparison, the systerll can check that JP is a prirnary key for table CJP by rnaintaining an index on JP. This difference in integrity-checking cost is the rl1otivation for dependency-preservation. On the other hand, if updates are infrequent, this increased cost IIlay be acceptable; therefore, we rnight choose not to rnaintain the table C.JP (and quite likely, an index all it).

As another exarnple illustrating decornposition choices, consider the Contracts relation again, and suppose that we also have the integrity constraint that a departrnent uses a given supplier for at rnost one of its projects: $SPQ \rightarrow V$. Proceeding as before, we have a lossless-join decornposition of Contracts into SDP and CSJDQV. Alternativcly, we could begin by using the dependency $SPQ \rightarrow V$ to guide our decomposition, and replace Contracts with SPQV and CSJDPQ. We can then dec(Hnpose CSJDPQ, guided by $51D \rightarrow P$, to obtain SDP and CSJDQ.

We now have two alternative lossless-join decornpositions of Contracts into BCNF, neither of which is dependency-preserving. The first alternative is to replace Contracts with the relations SDP and CSJDQV. The second alternative is to replace it \vith SPQV, SDP, and CSJDQ. The addition of CJP makes the second deC0111positioll (but not the first) dependency-preserving. Again, the cost of lnaintaining the three relations CJP, SPQV, and CSJDQ (versus just CSJDQV) Illay lead us to choose the first alternative. In this case, enforcing the given FDs becornes Inore expensive. We Illight consider Ilot enforcing thern, but we then risk a violation of the integrity of our data.

20.8.4 Vertical Partitioning of BCNF Relations

Suppose that we have decided to decornpose Contracts into SDP and CSJDQV. These scheruas are in BCNF, and there is no reason to decornpose thern further from a nonl1alization standpoint. However, suppose that the following queries are very frequent:

- Find the contracts held by supplier S.
- Find the contracts placed by departrnent D.

These queries rnight lead us to decompose CSJDQV into CS, CD, and CJQV. The decornposition is lossless, of course, and the two il11portant queries can be answered by exarnining 111uch slualler relations. Another reason to consider such a dec()l11position is concurrency control *hot spots*. If these queries are COIllIllon, and the rnost COIlunon updates involve changing the quantity of products (and the value) involved in contracts, the decoulposition inlproves perforrnance by reducing lock contention. Exclusive locks are now set rnostly on the CJQV table, and reads on CS and CD do not conflict with these locks.

Whenever we decornpose a relation, we have to consider which queries the decolnposition rnight adversely affect, especially if the only rnotivation for the decoInposition is iUlproved perforrnance. For exaruplc, if another illlportant query is to find the total value of contracts held by a supplier, it would involve a join of the decornposed relations CS and CJQV. In this situation, we rnight decide against the decolnposition.

20.8.5 Horizontal Decomposition

Thus far, we have essentially considered how to replace a relation with a collection of vertical decorupositions. Sornetilnes, it is worth considering whether to replace a relation with two relations that have the sa.Dle attributes as the original relation, each containin.g a subset of the tuples in tb.e original. Intuitively, this technique is useful \vhen different subsets of tuples are queried in very distinct ways.

For example, different rules lnay govern large contracts, \vhich are defined as contracts with values greater than 10,000. (Perhaps, such contracts have to be awarded through a bidding process.) This constraint could lead to a nUlnber of queries in which Contracts tuples are selected using a condition of the forIII *value* > 10,000. On8 way to approach this situation is to build a clustered B+ tree index oII the *value* field of Contracts. Alternatively, we could replace Contracts with two relations called LargeContracts and SrnallContracts, with the obvious 11leaning. If this query is the only lllotivation for the index, horizontal decornposition offers all the benefits of the index without the overhead of index maintenance. This alternative is especially attractive if other irnportant queries on Contracts also require clustered indexes (on fields other than *val'ue).*

If we replace Contracts by two relations LargeContracts and SrnallContracts, we could r1l8sk this change by defining a view called Contracts:

```
CREATE VIEW Contracts(cid, supplierid, projectid, deptid, partid, qty, value)
       AS ((SELECT *
            FROM   LargeContracts)
            UNION
           (SELECT *
            FROM   SmallContracts))
```

However, any query that deals solely with LargeContracts should be expressed directly on LargeContracts and not on the view. Expressing the query on the view Contracts with the selection condition *value*> 10,000 is equivalent to expressing the query on LargeContracts but less efficient. This point is quite general: Although we can mask changes to the conceptual scherlla by adding view definitions, users concerned about perforrnance have to be aware of the change.

As another exanlple, if Contracts had an additional field *year* and queries typically dealt with the contracts in sorne one year, we rnight choose to pa,rtition Contracts by year. ()f course, queries that involved contracts fron1 rnore than one year rnight require us to pose queries against each of the decolllposed relations.

20.9 CHOICES IN TUNING QUERIES AND VIEWS

T'he first step in tuning a query is to understand the plan used by the DBMS to evaluate the query. 8ysten1s usually provide sorne facility for identifying the plan used to evaluate a query. Once we understand the plan selected by the systelIl, we can consider how to irnprove performance. We can consider a different choice of ilHlexes or perhaps co-clustering two relations for join queries, guided by our understanding of the old plan and a better plan that we want theDBlVIS to use. The details are sinlilar to the initial design process.

One point'worth rnaking is that before creating new indexes we should consider \vhether rewriting the query achieves acceptable results with existing indexes. For example, consider the foll()\ving query with an OR connective:

```
SELECT  E.dno
FROM    Ernployees E
WHERE   E.hobby='Stalnps' OR E.age==10
```

If \ve have indexes on both *hobby* and *age*, we can use these indexes to retrieve the necessary tuples, but an optiInizer ruight fail to recognize this opportunity. The optinlizer rnight view the conditions in the WHERE clause as a whole as not rnatching either index, do a sequential scan of Ernployees, and apply the selections on-the-fly. Suppose we rewrite the query as the union of two queries, one with the clause *WHEREE.hobby='Starnps"* and the other with the clause WHERE *E.agc==10.* Now each query is answered efficiently with the aid of the indexes on *hobby* and *age*.

We should also consider rewriting the query to avoid sorne expensive operations. For exalnple, including DISTINCT in the SELECT clause leads to duplicate elirn-ination, which can be costly. rrhus, we should ornit DISTINCT whenever possible. For exalnple, for a query on a single relation, we can ornit DISTINCT whenever either of the following conditions holds:

- We do not care about the presence of duplicates.
- rrhe attributes lllentioned in the SELECT clause include a candidate key for the relation.

SOlnetirnes a query with GROUP BY and HAVING can be replaced by a query without these clauses, thereby eliminating a sort operation. For example, consider:

```
SELECT    MIN (E.age)
FROM      Employees E
GROUP BY  E.dno
HAVING    E.dno=102
```

This quer:y is equivalent to

```
SELECT    MIN (E.age)
FROM      Erl1ployees E
WHERE     E.dno=102
```

Cornplex queries are often written in steps, using a temporary relation. We can usually rewrite such queries without the tClnporary relation to rnake thcrn run faster. Consider the following query for cornputi.ng the average salary of departrnents rnanaged by Robinson:

```
SELECT    *
INTO      Ternp
FROM      ErnployeesE, Depa:rtruents D
WHERE     E.dno=D.dno AND D.rngrnanle='Robinson'

SELECT    T.dno, AVG (T.sal)
FROM      T'clnp T
GROUP BY  T.dno
```

This query can be rewritten as

```
SELECT    E.dno, AVG (E.sal)
FROM      Elnployees E, Departments D
WHERE     E.dno==D.dno AND D.rngrnarne=='llobinson'
GROUP BY  E.dno
```

The rewritten query does not 111aterialize the interrnediate relation ,Ternp and is therefore likely to be faster. In fact, the optimizer may even find a very efficient index-only plan that never retrieves Ernployees tuples if there is a cornposite B+· tree index on *(d'no, sal).* This exanlple illustrates a general observation: *By rewriting queries to avoid unnecessary temporaries, we not only avoid creating the ternporary relations, we also open up rnore optimization possibilities for the optim,izer to el;plore.*

In SCHne situations, ho\vever, if the optirnizer is unable to find a good plan for a cornplex query (typically a nested query with correlation), it rnay be worthwhile to rewrite the query using tenlporary relations to guide the optirnizer toward a good plan.

In fact, nested queries are a conunon source of inefficiency because luany optirnizers deal poorly with theIn, as discussed in Section 15.5.v'Vllenever possible, it is better to rewrite a nested query \vithout nesting and a correlated query without correlation. As already noted, a good reforrIlulation of the query rnay require us to introduce new, ternporary relations, and techniques to do so systenlatically (ideally, to be done by the optirnizer) have been \videly studied. ()ften tllough, it is possible to re\vrite nested queries ,vithout nesting or the use of ternpora,ry relations, as illustrated in Section 15.5.

20.10 IMPACT OF CONCURRENCY

In a system with IllalIY concurrent users, several additional points IllUSt be considered. Transactions obtain *locks* on the pages they access, and other transactions Ina)' be blocked waiting for locks on objects they wish to access.

We observed in Section **16.5** that blocking delays 111ust be IniniInized for good perforrnance and identified two specific ways to reduce blocking:

- Reducing the time that transactions hold locks.
- R,edllcing hot spots.

We now discuss techniques for achieving these goals.

20.10.1 Reducing Lock Durations

Delay Lock Requests: Tune transactions by writing to local prograrn variables and deferring changes to the database until the end of the transaction. This delays the acquisition of the corresponding locks and reduces the time the locks are held.

Make Transactions Faster: The sooner a transaction c01npletes, the sooner its locks are released. We have already discussed several ways to speed up queries and updates (e.g., tUllillg indexes, rewriting queries). In addition, a careful partitioning of the tuples in a relation and its associated indexes across a collection of disks can significantly irnprove concurrent access. For exarnple, if we have the relation on one disk and an index on another, accesses to the index can proceed without interfering with accesses to the relation, at least at the level of disk reads.

Replace Long Transactions by Short Ones: SometiInes, just too ruuch work is done within a transaction, and it takes a long tirne and holds locks a long tirne. Consider rewriting the transaction as two or Inore smaller transactions; holdable cursors (see Section 6.1.2) can be helpful in doing this. The advantage is that each new transaction cornpletes quicker and releases locks sooner. The disadvantage is that the original list of operations is no longer executed atomically, and the application code Illust deal with situations in which one or rnore of the new transactions fail.

Build a Warehouse: CC)lnplex queries can hold shared locks for a long tirne. ()ften, however, these queries involve statistical analysis of business trends and it is a,cceptable to run theln on a copy of the data that is a little out of date. rrhis led to the popularity of *data warehouses*, which are databases that cornplcnlcnt the operational database by rnaintaining a copy of data used in cornplex queries (Chapter 25). Running these queries against the warehouse relieves the burden of long-running queries froln the operational database.

Consider a Lower Isolation Level: In rnany situations, such as queries generating aggregate infonnation or statistical sununaries, we can use a lower SQL isolation level such as REPEATABLE READ or READ COMMITTED (Section 16.6). Lo\ver isolation levels incur lower locking overheads, a,nel the application programnner rllust make good design trade-offs.

20.10.2 **Reducing Hot Spots**

Delay Operations on Hot Spots: We already discussed the value of delaying lock requests. Obviously, this is especially irnportant for requests involving frequently used objects.

Optimize Access Patterns: The *patteTn* of updates to a relation can also be significant. For exanlple, if tuples are inserted into the Ernployees relation in *eid* order and we have a B+ tree index on *eid*, each insert goes to the last leaf page of the B+ tree. This leads to hot spots along the path froIn the root to the rightrnost leaf page. Such considerations nlay lead us to choose a hash index over a B+- tree index or to index on a different field. Note that this pattern of access leads to poor perforrnance for ISAM indexes as well, since the last leaf page beCOlIles a hot spot. rrhis is not a problcln for hash indexes because the hashing process randornizes the bucket into which a record is inserted.

Partition Operations on Hot Spots: Consider a data entry transaction that appends new records to a file (e.g., inserts into a table stored as a heap file). Instead of appending records one-per-transaction and obtaining a lock on the last page for each record, we can replace the transaction by several other transactions, each of which writes records to a local file and periodically appends a batch of records to the rnain file. While we do rnore work overall, this reduces the lock contention on the last page of the original file.

As a further illustration of partitioning, suppose we track the nU111ber of records inserted in a counter. Instead of updating this counter once per record, the preceding approach results in updating several counters and periodically updating the main counter. rrhis idea can be adapted to many uses of counters, \vith varying degrees of effort. For exaInple, consider a counter that tracks the IIurnber of reservations, with the rule that a new reservation is allowed only if the counter is below a, rnaxiullun value. We can replace this by three counters, each \vith one-third the origina11naxirIlurn threshold, and three transactions that use these counters rather than the original. We obtain greater concurrency, but have to deal with the case where one of the counters is at the maximum value but SOHle other counter can still be incrcrnented. Thus, the price of greater concurrency is increased cornplexity in the logic of the application code.

Choice of Index: If a relation is updated frequently, B+ tree indexes can becolne a concurrency control bottleneck, because all accesses through the index HUlst go through the root. Thus, the root and index pages just below it can bec()lne hot spots. If the DBMS uses specialized locking protocols for tree indexes, and in particular, sets finc-granularity locks, this problenl is greatly alleviated. l\Ilany current systeuls use such techniques.

Nonetheless, this consideration lllay lead us to choose an ISAM index in SOllIe situations. Because the index levels of an ISAM index are static, we need not obtain locks on these pages; only the leaf pages need to be locked. An ISAl\!l index rnay be preferable to a B+ tree index, for exalllple, if frequent updates occur but we expect the relative distribution of records and the nUlnber (and size) of records with a given range of search key values to stay approxirnately the saIne. In this case the ISAM index offers a lower locking overhead (and reduced contention for locks), and the distribution of records is such that few overflow pages are created.

I-Iashed indexes do not create such a concurrency bottleneck, unless the data distribution is very skewed and lnany data itenlS are concentrated in a few buckets. In this case, the directory entries for these buckets can beccnne a hot spot.

20.11 CASE STUDY: THE INTERNET SHOP

Revisiting our running case study, I)BDudes considers the expected workload for the B&N 1)00kstore. rrhe owner of the bookstore expects rnost of his CIIStorners to search for books by ISBN nUluber before placing an order. Placing an order involves inserting one record into the ()rders table and inserting one or lllore records into the Orderlists relation. If a sufficient nurnber of books is available, a, shiprnent is prepared and a value for the *ship_date* in the Orderlists relation is set. In addition, the available quantities of books in stock changes all the tirne, since orders are placed that; decrease the quantity available and new books arrive frorn suppliers and increase the quantity available.

The DBDudes tearn begins by considering searches for books by ISBN'. Since *isbn* is a key, (l,n equality query on *isbn* returns at rnost one record. rrhereforc, to speed up queries frolll Cllstolllers who look for books with a given ISBN, I)BIJudes decides to build an unclustered hash index on *isbn*.

Next, it considers updates to book quantities. To update the *qty_in_stock* value for a book, we IIIUSt first search for the book by ISBN; the index on *isbn* speeds this up. Since the *qty_in_stock* value for a book is updated quite frequently, DBDudes also considers partitioning the Books relation vertically into the fo1-1c)\ving two relations:

BooksQty(<u>*isbn*</u>, *qty*)
BookRest(<u>*isbn*</u>, *title*, *author*, *price*, *year_published*)

Unfortunately, this vertical partitioning slows do\vn another very popular query: Equality search on ISBN to retrieve all infonnation about a book now requires a join between BooksQty and BooksH,est. So DBDudes decides not to vertically partition Books.

DBDudcs thinks it is likely that custonlers will also want to search for books by title and by author, and decides to add unclustered hash indexes on *title* and author-these indexes are inexpensive to rnaintain because the set of books is rarely changed even though the quantity in stock for a book changes often.

Next, DBDudes considers the Custorners relation. A custorner is first identified by the unique custorner identifaction nurnber. So the rnost COlnrnon queries on Custorners are equality queries involving the custolner identification nurnber, and DBDudes decides to build a clustered hash index on *cid* to achieve maxirnum speed for this query.

l\!Ioving on to the Orders relation, DBDudes sees that it is involved in two queries: insertion of new orders and retrieval of existing orders. Both queries involve the *ordernum* attribute as search key and so DBDudes decides to huild an index on it. What type of index should this be—a 13+ tree or a hash index? Since order nurnbers are assigned sequentially and correspond to the order date, sorting by *ordernum* effectively sorts by order date as well. So DBDudes decides to build a clustered B+ tree index on *ordernum*. A.lthough the operational requirernents rncntioned until no\v favor neither a 13+ tree nor a hash index, B&N\vill probably want to rnonitor daily a,ctivities and the clustered 13+ tree is a better choice for such range queries. ()f course, this 1118ans that retrieving all orders for a given custorner could be expensive for custolllers with InallY orders, since clustering by *ordernum* precludes clustering by other attributes, SllCh as *cicio*

The ()rderlists rela,tion involves lnostly insertions, with an occasional update of a shiprnent date or a query to list all cOlnponents of a given order. If Orderlists is kept sorted on *ordernum*, all insertions are appends at the end of the relation and thus very efficient. A clustered 13+ tree index on *ordernum* maintains this sort order and also speeds up retrieval of aU iterns for a given order. To update a shiprnent date, we need to search for a tuple by *ordernum* and *isbn*. The index on *ordernum* helps here as well. Although an index on ⟨*ordernum, isbn*⟩ would be better for this purpose, insertions would not be as efficient as with an index on just *ordernum*; DBDudes therefore decides to index ()rderlists on just *ordernum*.

20.11.1 Tuning the Database

Several rnonths after the launch of the B&N site, DBDudes is called in and told that custorner enquiries about pending orders are being processed very slowly. B&N has becorne very successful, and the Orders and Orderlists tables have grown huge.

l'hinking further about the design, DBDudes realizes that there are two types of orders: *completed orders,* for which all books have already shipped, and *partially co'mpleted orders*, for which sorne books are yet to be shipped. l\Ilost custorIler requests to look up an order involve partially corIlpleted orders, which are a sInall fraction of all orders. DBDudes therefore decides to horizontally partition both the Orders table and the Orderlists table by *ordernu'Tn*. This results in four new relations: NewOrders, OldOrders, NewOrderlists, and OldOrderlists.

An order and its cornponents are always in exactly one pair of relations—and we can deterrIline which pair, old or new, by a sinlple check on *ordernum*—and queries involving that order can always be evaluated using only the relevant relations. SCHIle queries are now slower, such as those asking for all of a customer's orders, since they require us to search two sets of relations. lIowever, these queries are infrequent and their perforrnance is acceptable.

20.12 DBMS BENCHMARKING

Thus far, we considered how to irnprove the design of a database to obtain better perforrnance. 1\s the database grows, however; the underlying DBMS rnay no longer be able to provide adequate perforrnance, even with the best possible design, and we have to consider upgrading our systcrn, typically by buying faster harchva,re and additional rnernory. We IIlay also consider rnigrating our database to (1, new DBIVIS.

When evaluating DBMS products, perforrnal1ce is an iUlportant consideration. ADBIVIS is a cornplex piece of software, and different vendors rnay target their systerns toward different market segrnents by putting rnore effort into optirnizirlg certa,in parts of the systern or choosing different systern designs. For example, sorne systcrIls are designed to run cornplex queries efficiently, while others are designed to run Inany sirnple transactions per second. Within each category of systcrIls, there are many cornpeting products. To assist users in choosing a DBMS that is 'well suited to their needs, several performance benchmarks have been developed. These include benchrnarks for measuring the perforlnance of a certain class of applications (e.g., the TPC benclnnarks) and benchrnarks for measuring how well a DBIVIS perfOrII1S various operations (e.g., the \Visconsin benchrnark).

Benchmarks should be portable, easy to understand, and scale naturally to larger problenl instances. They should II1eaSUre *peak performance* (e.g., *transactions per second,* or *ips)* as well as *pTice/perforrnance ratios* (e.g., *$/tps)* for typical workloads in a given application donlain. The Transaction Processing Council (TPC) was created to define benchlnarks for transaction processing and database systerns. Other well-known benchlnarks have been proposed by acadelnic researchers and industry organizations. Benchrnarks that are proprietary to a given vendor are not very useful for cornparing different systerns (although they rnay be useful in deterrnining how well a given systern would handle a particular workload).

20.12.1 Well-Known DBMS Benchmarks

Online Transaction Processing Benchmarks: The TPC-A and TPC-B benchrnarks constitute the standard definitions of the *ips* and $/*tps* measures. TPC-A rneasures the perfonnance and price of a computer network in addition to the DBMS, whereas the TPC-B benclnnark considers the DBMS by itself. These bencln11arks involve a sirnple transaction that updates three data records, frolIl three different tables, and appends a record to a fourth table. A 11urnber of details (e.g., transaction arrival distribution, interconnect rnethod, systern properties) are rigorously specified, ensuring that results for different systenls can be rneaningfully cOI11pared. The T'PC-C benchmark is a l110re cornplex suite of transactional tasks than TPC-A and TPC-B. It rnodels a warehouse that tracks iterns supplied to custorners and involves five types of transactions. Each TPC-C transaction is rnuch rIlore expensive than a 1'PC-A or TPC-B transaction, a,nel TPC-C exercises a rnuch ,videI' range of systern capabilities, such as use of secondary indexes and transaction aborts. It has Inore or less cOlnpletely replaced TPC-A and rrpC-B as the standard transaction processing bencillnark.

Query Benchmarks: The Wisconsin l)cnchrnark is \videly used for 1neasnring the perforrnance of sirnple relational queries. The Set Query benclunark Hleasures the perforrnance of a suite of rJlore cornplex queries, and the AS^3AP l)enchrnark measures the perfonnance of a Inixed workload of transactions, relational queries, and utility fUllctions. The rrpC-I) benchn.lark is a suite of cornplex SQL queries intended to be representative of the (Incision-support application domain. The ()LAP Council also developed a benchmark for cornplex decision-support queries, including some queries that cannot be expressed easily in SQL; this is intended to rneasure systerIls for *online a'nalyt'ic processing (OLAP)*,\vhieh we discuss in Chapter 25, rather than traditional SQL systerns. The Sequoia 2000 benchrnark is designed to cornpare DBMS support for geographic inforrnation systerns.

Object-Database Benchmarks: The 001 and 007 benclunarks rneasure the perforrnance of object-oriented database systelns. 'rhe Bucky benclunark rneasures the perforrnance of object-relational database systcrns. (We discuss object-database systelns in Chapter 23.)

20.12.2 Using a **Benchmark**

Benchrnarks should be used with a good understanding of what they are designed to rnea8ure and the application environrnent in \vhich a DBMS is to be used. When you use benchrnarks to guide your choice of a DBMS, keep the following guidelines in rnind:

- How Meaningful is a Given Benchmark? Benchrnarks that try to distill perforrnance into a single nunlber can be overly sirnplistic. A DBMS is a cOlnplex piece of software used in a variety of applications. A good benchlnark should have a suite of tasks that are carefully chosen to cover a particular application dornain and test DBMS features irnportant for that d01nain.
- How Well Does a Benchrnark Reflect Your Workload? Consider your expected workload and corupare it with the benchrnark. Give 11101'8 weight to the perfonnance of those l)enchrnark tasks (i.e., queries and updates) that are similar to irnportant tasks in your workload. Also consider how benclunark nurnbers are measured. For exarnple, elapsed tirne for individual queries rnight be rnisleading if considered in a rnultiuser setting: A systern may have higher elapsed times because of slower *l/C)*. On a 1nultiuser workloa,d, given sufficient disks for parallel *l/C)*, such a systern lnight olltperfofrn a SY8t8111 'with a lower elapsed time.
- Create Your Own Benchmark: Vendors often tweak their systerns in ad hoc ways to obtain good nurnbers on important benchmarks. fro counter this, create your own benc1unark by modifying standard benchrnarks slightly or by replacing the tasks in a standard benchrnark \vith similar tasks frarn your workload.

20.13 REVIEW QUESTIONS

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

- What are the cornponents of a workload description? (Section 20.1.1)
- What decisions need to be rnade during physical design? (Section 20.1.2)
- Describe six high-level guidelines for index selection. (Section 20.2)
- When should we create clustered indexes? (Section 20.4)
- What is co-clustering, and when should we use it? (Section 20.4.1)
- What is an index-only plan, and how do we create indexes for index-only plans? (Section 20.5)
- Why is automatic index tuning a hard problern? Give an exarnple. (Section 20.6.1)
- Give an exarnple of one algorithrn for autonlatic index tuning. (Section 20.6.2)
- Why is database tuning irnportant? (Section 20.7)
- How do we tune indexes, the conceptual scheula, and queries and views? (Sections 20.7.1 to 20.7.3)
- What are our choices in tuning the conceptual scherna? What are the following techniques and when should we apply thern: settling for a weaker norrnal forrn, denorrnalization, and horizontal and vertiacal decornpositions. (Section 20.8)
- What choices do we have in tuning queries and views? (Section 20.9)
- What is the irnpact of locking 011 database perforluance? I-Iow can we reduce lock contention and hot spots? (Section 20.10)
- Why do we have standaTdized database benclllnarks, and what conunon Inetrics are used to evaluate database systelns? Can you describe a few popular database benchrnarks? (Section 20.12)

EXERCISES

Exercise 20.1 Consider the following BCNF schcrna for a portion of a sirnple corporate database (type infonnation is not relevant to this question and is ornitted):

> Ernp (*eid*, *ename*, *addr*, *sal*, *age*, *yrs*, *deptid*)
> Dept (*did*, *dname*, *flooT*, *budget*)

Suppose you know that the following queries are the six most COUUIlcm queries in the \vorkload for this corporation and that all six are roughly equivalent in frequency and inlportance:

- List the id, name, and address of eUlployees in a user-specified age range.
- List the id, name, and address of crnployees who work in the department \vith a llser-specified department narne.
- List the id and address of elnployees with a user-specified eluployeenanle.
- List the overall average salary for ernployees.
- List the average salary for eInployees of each age; that is, for each age in the datal)(1se, list the age and the corresponding average salary.
- List all the departrnent infonnation, ordered by departrnent floor nurnbers.

1. Given this infonnation, and assuIning that these queries are lnore iluportant than any updates, design a physical scherna for the corporate database that will give good perfor-rnance for the expected workload. In particular, decide which attributes will be indexed and whether each index will be a clustered index or an unclustered index. Assume that 13+ tree indexes are the only index type supported by the DBMS and that both single- and nnrltiple-attribute keys are pernlitted. Specify yOllr physical design by identifying the attributes you recornnlCnd indexing on via clustered or unclustered 13+ trees.
2. Redesign the physical schelna assuIning that the set of iInportant queries is changed to be the following:
   - List the id and address of enlployees with a user-specified ernployee narne.
   - List the overall rnaxinHun salary for eruployees.
   - List the average salary for ernployees by departnlent; that is, for each *deptid* value, list the *deptid* value and the average salary of ernployees in that departrnent.
   - List the Slun of the budgets of all departrnents by floor; that is, for each floor, list the floor and the sum.
   - AssuIne that this workload is to be tuned with an autornatic index tuning wizard. Outline the rnain steps in the execution of the index tuning algorithrn and the set of candidate configurations that would be considered.

Exercise 20.2 Consider the follo\ving BCNF' relational scherna for a portion of a universit.y database (type infonnation is not relevant to this question and is ornitted):

Prof(<u>*ssno*</u>, *pname*, office, *age*, *sex*, *specialty*, *dept_did*)
Dept(<u>*did*</u>, *dname*, *budget*, *num_majors*, *chair_ssno*)

Suppose you kno\v that the following queries are the five rnost connnon queries in the workloa,d for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and offices of professors of a user-specified sex (rnale or fernale) who have a user-specified research specialty (e.g., *recursive query processing*). Assurne that the university has a diverse set of faculty members, rnaking it very unCOlnmon for Inore than a fe\\! professors to have the same research specialty.
- List all the departrnent information for departrnents with professors in a user-specified age range.
- List the <lepartlnent i<l, department name, and chairperson name for departments with a user-specified nurnber of majors.

- List the lowest budget for a department in the university.
- List all the infornultion about professors \vho are departlnent chairpersons.

These queries occur runch 1nore frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schc1na for the university database that will give good perfonnance for the expected workload. In particular, decide which attributes should be indexed and 'whether each index should be a clustered index or an unclustered index. Assulne that both B+ trees and hashed indexes are supported by the DBMS and that both single- and lImltiple-attribute index search keys are perrnitted.

1. Specify your physical design by identifying the attributes you recomlnend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Assurne that this workload is to be tuned with an autornatic index tuning wizard. Outline the rnain steps in the algorithrn and the set of candidate configurations considered.
3. Redesign the physical schema, assurning that the set of irnportant queries is changed to be the following:
   - List the nUlnber of different specialties covered by professors in each department, by department.
   - Find the departrnent with the fewest rnajors.
   - Find the youngest professor who is a department chairperson.

Exercise 20.3 Consider the following BCNF relational schmna for a portion of a cornpany database (type inforrnation is not relevant to this question and is OInitted):

> Project*(pno, proj_name, proj_base_dept, proj_mgr, topic, budget)*
> Manager(*mid, mgr_name, mgr_dept, salary, age, sex:)*

Note that each project is based in sorne department, each manager is e1Ylployed in some departIllEmt, and the manager of a project need not be e1nployed in the sarne departrnent (in which the project is based). Suppose you know that the following queries are the five most COHUllon queries in the workload for this university and all five are roughly equivalent in frequency and i1"nportance:

- List the names, ages, and salaries of lnanagers of a user-specified sex (rnale or female) working in a given department. You can assurne that, while there are rnany departments, each department contains very few project managers.
- List the names of all projects with lnanagers whose ages are m a user-specified range (e.g., younger than 30).
- List the names of all departrnents such that a rnanager III this department manages a project based in this department.
- List the name of the project with the lowest budget.
- List the names of all managers in the saITIC department as a given project.

These queries occur nIllch more frequently than updates, so you should build \vhatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates \'lill occur (a"nd \vould be slowed down by urmccessary indexes). Given this infonnatioll, design a physical schema for the conlpany database that win give good performance for the expected \vorkload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assuille that both B+ trees and hashed indexes are supported by the DBMS, and that both single- and Illuitiple-attribute index keys are perruitted.

1. Specify your physical design by identifying the attributes you recOlIlrIlend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Assunle that this workload is to be tuned with an autornatic index tuning wizard. Outline the lnain steps in the algorithrn and the set of candidate configurations considered.
3. Redesign the physical schema assulning the set of ilnportant queries is changed to be the following:
   - Find the total of the budgets for projects luanaged by each rnanager; that is, list *p'roj_rngr* and the total of the budgets of projects luanaged by that manager, for all values of *proj_mgT.*
   - Find the total of the budgets for projects managed by each rnanager but only for managers who are in a user-specified age range.
   - Find the number of male rnanagers.
   - Find the average age of rnanagers.

Exercise 20.4 The Globetrotters Club is organized into chapters. The president of a chapter can never serve as the president of any other chapter, and each chapter gives its president sonle salary. Chapters keep moving to new locations, and a new president is elected when (and only when) a chapter rnoves. This data is stored in a relation *G(C,S,L,P)*, where the attributes are chapters *(C)*, salaries (S), locations (L), and presidents *(P)*. Queries of the following fornl are frequently asked, and you *mU8t* be able to answer thern without cOluputing a join: "Who was the president of chapter X when it was in location Y?"

1. List the FDs that are given to hold over G.
2. What are the candidate keys for relation G?
3. What Honnal fornl is the scherna Gin?
4. Design a good database scherna for the club. (Rernernber that your design *must* satisfy the stated query requirenlent!)
5. What nonnal fonn is your good scherna in? Give an exarnple of a query that is likely to run slower on this schema than on the relation G.
6. Is there a lossless-join, dependency-preserving deCOlTlposition of G into BeNF?
7. Is there ever a good reason to accept sornething less than :3NF' \vhen designing a schema for a relational database? Use this example, if necessary adding further constraints, to illustrate your answer.

Exercise 20.5 Consider the following BCNF relation, which lists the ids, types (e.g., nuts or bolts), and costs of various parts, along with the mllnber available or in stock:

Parts (<u>*pid*</u>, *pname, cost, num_avail*)

You are told that the following two queries are extrelnely irnportant:

- Find the total nunlber available by part type, for all types. (That is, the surn of the *num_avail* value of all nuts, the sum of the *num_avail* value of all bolts, and so forth)
- List the *pids* of parts with the highest cost.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for the set of Parts records, and what indexes would you create?
2. Suppose your custorners subsequently cmnplain that performance is still not satisfactory (given the indexes and file organization you chose for the Parts relation in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schenla redesign. Explain how you would try to obtain better perfonnance by describing the scherna for the relation(s) that you would use and your choice of file organizations and indexes on these relations.
3. How would your answers to the two questions change, if at all, if your systeIl1 did not support indexes with multiple-attribute search keys?

Exercise 20.6 Consider the following BCNF relations, which describe ernployees and the departments they work in:

Ernp (*eid*, *sal*, *did)*
Dept *(d'id*, *location*, *budget*)

You are told that the following queries are extrernely important:

- Find the location where a user-specified enlployee works.
- Check whether the budget of a department is greater than the salary of each ernployee in that departrnent.

1. Describe the physical design you would choose for this relation. That is, what kind of a file structure would you choose for these relations, and what indexes would you create?
2. Suppose that your custollwrs subsequently cOIuplain that perforrnance is still not satisfactory (given the indexes and file organization that you chose for the relations in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schelna redesign. Explain how you would try to obtain better perfonnance by describing the scherna for the relation(s) that you would use and your choice of file organizations and indexes on these relations.
3. Suppose that your database systern has very ineff1cient irnplenlentations of index structures. What kind of a design would you try in this case?

Exercise 20.7 Consider the following BCNF relations, which describe departrnents in a company and ernployees:

Dept(*did*, *dname*. *location* *managerid*)
Enlp(*eid*, *sal*)

'You are told that the followving queries are extrernely iruportant:

- List the names and ids of rnanagel's for each department in a user-specified location., in alphabetical order by department narne.
- Find the average salary of ernployees who manage departments in a user-specified location.You can assume that no one rnanages nlOre than one depa,rtrnent.

1. Describe the file structures and indexes that you would choose.
2. You subsequently realize that updates to these relations are frequent. Because indexes incur a high overhead, can you think of a way to irnprove perforrnance on these queries without using indexes?

Exercise 20.8 For each of the following queries, identify one possible reason why an opti-Inizer Illight not find a good plan. Rewrite the query so that a good plan is likely to be found. Any available indexes or known constraints are listed before each query; assurne that the relation schelnas are consistent with the attributes referred to in the query.

1. An index is available on the *age* attribute:

```
SELECT  E.dno
FROM    Elnployee E
WHERE   E.age=20 OR E.age=10
```

2. A B+ tree index is available on the *age* attribute:

```
SELECT  E.dno
FROM    Employee E
WHERE   E.age<20 AND E.age>10
```

3. An index is available on the *age* attribute:

```
SELECT  E.eIno
FROM    Enlployee E
WHERE   2*E.age<20
```

4. No index is available:

```
SELECT  DISTINCT *
FROM    Enlployee E
```

5. No index is available:

```
SELECT    AVG (B.sal)
FROM      Elnployee E
GROUP BY  E.dno
HAVING    E.dno=22
```

6. The *sid* in Reserves is a foreign key that refers to Sailors:

```
SELECT    S.sid
FROM      Sailors S, Reserves H
WHERE     S.sid=R.sid
```

Exercise 20.9 Consider two 'ways to COlupute the names of elnployees who earn rnore than $100,000 and whose age is equal to their manager's age. First, a nested query:

```
SELECT   E1.ename
FROM     Emp E1
WHERE    El.sal > 100 AND E1.age = ( SELECT  E2.age
                                     FROM    Ernp E2, Dept D2
                                     WHERE   E1.dname = D2.dname
                                             AND D2.mgr = E2.enarne )
```

Second, a query that uses a view definition:

```
SELECT   £1.enarne
FROM     Ernp E1, MgrAge A
WHERE    El.dnarue = A.dnarne AND E1.sal > 100 AND E1.age = A.age

CREATE VIEW MgrAge (dnan1e, age)
       AS SELECT D.dnanw, E.age
          FROM   Errlp E, Dept D
          WHERE  D.mgr = E.ename
```

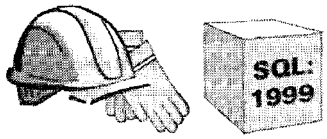
1. Describe a situation in which the first query is likely to outperforrn the second query.
2. Describe a situation in which the second query is likely to outperfonn the first query.
3. Can you construct an equivalent query that is likely to beat both these queries when every ernployee who earns rnore than $100,000 is either 35 or 40 years old? Explain briefly.

BIBLIOGRAPHIC NOTES

[658] is an early discussion of physical database design. [659] discusses the performance implications of normalization and observes that denormalization may improve perforrnance for certain queries. The ideas underlying a physical design tool frorn IBl'vf are described in [272]. The Microsoft AutoAdrnin tool that perfonns automatic index selection according to a query workload is described in several papers [163, 164]. The DB2 Advisor is described in [750]. Other approaches to physical database design are described in [146, 639]. [679] considers *transaction tuning*, which we discussed only briefly. The issue is how an application should be structured into a collection of transactions to rnaxirnize perfonnance.

The following books on database design cover physical design issues in detail; they are reCOIll-rnended for further reading. [274] is largely independent of specific products, although rnaBy examples are based on DB2 and Teradata systerl1S. [779] deals prirnarily with DB2. Shasha and Bonnet give an in-depth, readable introduction to database tuning [104].

[334] contains several papers on benchrnarking database systerns and has accompanying software. It includes articles on the AS^3AP, Set Query, 'I'PC-A, 'rpC-B, Wisconsin, and 001 bendunarks written by the original developers. The Bucky benchrnark is described in [132], the 007 benchrnark is described in [131], and the T'pe-D benchrnark is described in [739]. The Sequoia 2000 bendunark is described in [720].



21

SECURITY AND AUTHORIZATION

- What are the rnain security considerations in designing a database application?
- What IIlechanisms does a DBMS provide to control a user's access to data?
- What is discretionary access control and how is it supported in SQL?
- What are the weaknesses of discretionary access control? How are these addressed in lnandatory access control?
- What are covert channels and how do they cornpromise lnandatory access control?
- What lTIUst the DBA do to ensure security?
- What is the added security threat when a database is accessed re-rnotely?
- What is the role of encryption in ensuring secure access? How is it used for certifying servers and creating digital sig11atures?

- **Key concepts:** security, integrity, availability; discretionary access control, privileges, GRANT, REVOKE; rna.ndatory access control, objects, subjects, security classes, rnultilevel tables, polyinstantiation; covert channels, DoD security levels; statistical databases, inferring secure information; authentication for reIllote access, securing servers, digital signatures; encyption, public-key encryption. -

I know that's a secret, for it's whispered everywhere.

—*William Congreve*

The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset. In addition to protecting the intrinsic value of the data, corporations rnust consider O\vays to ensure privacy and control access to data that must not be revealed to certain groups of users for various reasons.

In this chapter, we discuss the concepts underlying access control and security in a DBMS. After introducing database security issues in Section 21.1,we consider two distinct approaches, called *discretionary* and *mandatory*, to specifying and lTlanaging access controls. An access control Inechanism is a way to control the data accessible by a given user. After introducing access controls in Section 21.2, we cover discretionary access control,which is supported in SQL, in Section 21.3.vVe briefly cover n1andatory access control, which is not supported in SQL, in Section 21.4.

In Section 21.6, we discuss SOlne additional aspects of database security, such as security in a statistical database and the role of the database adrninistrator. We then consider SOlne of the unique challenges in supporting secure access to a DBMS over the Internet, which is a central problern in e-COlllInerce and other Internet database applications, in Section 21.5. We conclude this chapter with a discussion of security aspects of the Barns and Nobble case study in Section 21.7.

21.1 INTRODUCTION TO DATABASE SECURITY

There are three rnain objectives when designing a secure database application:

1. Secrecy: InfoI'rnation should not be disclosed to unauthorized users. For exarnple, a student should not be allowed to exarnine other students' grades.
2. Integrity: ()nly authorized users should be allowed to Hlodify data. For example, students 1Ilay be allowed to see their grades, yet not allowed (obviously) to rnodify thern.
3. Availability: Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.

To achieve these objectives, a clear and consistent security policy should be developed to describe what security measures rnust be enforced. In particular, we rnu8t detennine what part of the data is to be protected and which users get access to which portions of the data. Next, the security mechanisrns of the underlying I)B:JVIS and operating systenl, as well as externaJ mechanisms, such as securing access to buildings, Illust be utilized to enforce the policy. We emphasize that security rneasures IIlust l)e taken at several levels.

Security leaks in the OS or network connections can cirCUlnvent database security rnechanisrns. For example, such leaks could allow an intruder to log on as the database acbninistrator, 'with all the attendant I)BlVIS access rights. Hurnan factors are another source of security leaks. :For exarnple, a user IHay choose a password that is easy to guess, or a user who is authorized to see sensitive data may luisuse it. Such errors account for a large percentage of security breaches. \Ve do not discuss these aspects of security despite their irllportance because they are not specific to database rnanagerllent systelIls; our IIlain focus is on database access control rllechanisrns to support a security policy.

We observe that vie\vs are a valuable tool in enforcing security policies. The view rnechanisrll can be used to create a 'window' 011 a collection of data that is appropriate for SOllIe group of users. 'Views allow us to liUlit access to sensitive data by providing access to a restricted version (defined through a view) of that data, rather than to the data itself.

We use the following schemas in our exaurples:

Sailors(*sid:* integer, *snarne:* string, *rating:* integer, *age:* real)
Boats(*bid:* integer, *bnarne:* string, *color:* string)
Reserves(*sid:* integer, *bid:* integer, *day:* dates)

Increasingly, as database systcrlls becorne the backbone of e-COlluncrce applications requests originate over the Internet. This rnakes it irnportant to be able to **authenticate** a user to the database systern. A.fter all, enforcing a security policy that allows user Sarn to read a table and Ehner to write the table is not of l11uch use if Saın can rnasquerade as Ebner. COllversely, we Inus!; be able to assure users that they a,re COIluIlunicating \vith a legitilnate systern (e.g., the real Arnazoll.col11 server, and not a spurious application intended to steal sensitive inforrnation such as a credit card nurl11>cr). vVhile the details of authentication are outside the scope of our coverage, we discuss the role of authentication and the l)Hsic ideas involved in Section 21.5, after covering database access control rnechanisrIls.

21.2 ACCESS CONTROL

A database for an enterprise contains a great deal of inforrnation and usually has several groups of users. 1\IJost users need to access onl,y a sruall pa;rt of the database to carry out their tasks. J\l1owing users unrestricted access to all the data can be undesirable, and a DBMS should provide rnechanisHls to control access to data.

A DBMS offers two rnain approaches to access control. Discretionary access control is based on the concept of access rights, or privileges, and rnechanisrllS for giving users such privileges. A privilege allows a user to access Borne data object in a certain IIlHnIler (e.g., to read or 11lOdify). A user who creates a database object such as a table or a view autornatically gets all applicable privileges on that object. The D.BMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all tirnes only users with the necessary privileges can access all object. SQL supports discretionary access control through the GRANT and REVOKE conunands. The GRANT cOllnnand gives privileges to users, and the REVOKE cornrnand takes away privileges. We discuss discretionary access control in Section 21.3.

Discretionary access control rnechanisrns, while generally effective, have certain weaknesses. In particular, a devious unauthorized user can trick an authorized user into disclosing sensitive data. Mandatory access control is based on systemwide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class,* each user is assigned *clearance* for a security class, and rules are irnposed on reading and writing of database objects by users. The DBMS deterrnines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be 'passed on' to a user without the necessary clearance. 'rhe SQL standard does not include any support for rnandatory access control. 'We discuss rnandatory access control in Section 21.4.

21.3 DISCRETIONARY ACCESS CONTROL

SQL supports discretionary access control through the GRANT and REVOKE cornrnands. The GRANT cornrnand gives users privileges to base tables and views. 'rhe syntax of this command is as follows:

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

For our purposes object is either a base table or a view. SQL recognizes certain other kinds of objects, but we do not discuss thcrn. Several privileges can be specified, including these:

- SELECT: The right to access (read) all colurnns of the table specified as the object, *including columns added later* through ALTER TABLE cornrnands.

- INSERT(*column-name*): The right to insert rows with (non-*null* or non-default) values in the naTned cohnnn of the table named as **object**. If this right is to be granted with respect to all cohunns, including coluulns that rnight be added later, we can sirnply usc INSERT. The privileges UPDATE(*column-name*) and UPDATE are sirnilar.
- DELETE: The right to delete rows frorn the table narned as **object**.
- REFERENCES(*col'Urnn-namJe):* The right to define foreign keys (in other tables) that refer to the specified cohnnn of the table **object**. REFERENCES without a colurnn naUIe specified denotes this right with respect to all colurnns, including any that are added later.

If a user has a privilege with the **grant option**, he or she can pass it to another user (with or without the grant option) by using the GRANT conunand. A user who creates a base table autolnatically has all applicable privileges on it, along with the right to grant these privileges to other users. A user who creates a view has precisely those privileges on the view that he or she has on *everyone* of the views or base tables used to define the view. The user creating the view Inust have the SELECT privilege on each underlying table, of course, and so is always granted the SELECT privilege on the view. The creator of the view has the SELECT privilege with the grant option only if he or she has the SELECT privilege with the grant option on every underlying table. In addition, if the view is updatable and the user holds INSERT, DELETE, or UPDATE privileges (with or without the grant option) on the (single) underlying table, the user autornatically gets the same privileges on the view.

()nly the owner of a scherna can execute the data definition statcrnents CREATE, ALTER, and DROP on that schcrna. The right to execute these staternents cannot be granted or revoked.

In conjullction with the GRANT and REVOKE cOl1llnands, views are an irnportant cornponent of the security rnechanisrns provided by a relational DBMS. By defining views on the base tables, we can present needed inforrnation to a user "while *hiding* other inforrnation that the user should not be given access to. For example, consider the following view definition:

```
CREATE VIEW }\ctiveSajlors (name, age, day)
       AS SELECT S.snarne, S.age, R.day
          FROM   Sailors S, Reserves R
          WHERE  S.sid = R.sid AND S.rating > 6
```

A user who can access ActiveSailors but not Sailors or Reserves kno\vs the names of sailors who have reservations but cannot find out the *bids* of boats reserved by a given sailor.

Role-Based Authorization in SQL: Privileges are assigned to users (authorization IDs, to be precise) in SQL-92. In the real world, privileges are often associated with a user's job or *role* within the organizat;ion. Many DBMSs have long supported the concept of a role and allowed privileges to be assigned to roles. Roles can then he granted to users and other roles. (Of courses, privileges can also be granted directly to users.) l'he SQL:1999 standard includes support for roles. Roles can be created and destroyed using the CREATE ROLE and DROP ROLE eornrnands. Users can be granted roles (optionally, with the ability to pass the role on to others). The standard GRANT and REVOKE connnands can assign privileges to (and revoke from) roles or authorization IDs.
What is the benefit of including a feature that Inany systerns already support? 'This ensures that, over tiIne, all vendors who comply with the standard support this feature. 'rhus, users can use the feature without worrying about portability of their application across DBMSs.

Privileges are assigned in SQL to authorization IDs, which can denote a single user or a group of users; a user lllUSt specify an authorization ID and, in Inany systerns, a corresponding *password* before the DBMS accepts any c0111-rnancls from hirn or her. So, technically, *Joe*, *Michael*, and so on are authorization IDs rather than user nan1es in the following exalllpies.

Suppose that user Joe has created the tables Boats, Reserves, and Sailors. Senne exarnples of the GRANT cOllunand that Joe can now execute fo11o\v:

```
GRANT INSERT, DELETE ON Reserves TO Yuppy WITH GRANT OPTION
GRANT SELECT ON Reserves TO Michael
GRANT SELECT ON Sailors TO Michael WITH GRANT OPTION
GRANT UPDATE (rating) ON Sailors TO Leah
GRANT REFERENCES (bid) ON Boats TO Bill
```

Yuppy can insert or delete Reserves rows and authorize SOlneone else to do the sarne. I\1ichael can execute SELECT queries on Sailors and H,eserves, and 118 can pass this privilege to others for Sailors but not for R,eserves. With the SELECT privilege, Michael can create a view that accesses the Sailors and Reserves tables (for example, the ActiveSailors vic\v), but he cannot grant SELECT on ActiveSailors to others.

()rl the other hand, suppose that Michael creates the foUo\ving view:

```
CREATE VIEWYoungSailors (sicl, age, rating)
       AS SELECT S.sicl, S.age, S.rating
```

```
FROM    Sailors S
WHERE   S.age < 18
```

The only underlying table is Sailors, for which Michael has SELECT with the grant option. He therefore has SELECT with the grant option on YoungSailors and can pass on the SELECT privilege on YoungSailors to Eric and Guppy:

```
GRANT SELECT ON YoungSailors TO Eric, Guppy
```

Eric and Guppy can now execute SELECT queries on the view YoungSailors-note, however, that Eric and Guppy do *not* have the right to execute SELECT queries directly on the underlying Sailors table.

Michael can also define constraints based on the inforrnation in the Sailors and Reserves tables. For exarnple, Michael can define the following table, which has an associated table constraint:

```
CREATE TABLE Sneaky (lnaxrating  INTEGER,
                     CHECK (maxrating >=
                            ( SELECT MAX (S.rating)
                              FROM   Sailors S )))
```

By repeatedly inserting rows with gradually increasing *rnaxrating* values into the Sneaky table until an insertion finally succeeds, lVIichael can find out the highest *rating* value in the Sailors table. This exarnple illustrates why SQL requires the creator of a table constraint that refers to Sailors to possess the SELECT privilege on Sailors.

Returning to the privileges granted by Joe, Leah can update only the *rating* colulnn of Sailors rows. She can execute the following cornmand, which sets all ratings to 8:

```
UPDATE  Sailors S
SET     S.rating = 8
```

However, she cannot execute the same cOllunand if the SET clause is changed to be SET *S.age* = 25, because she is not allowed to update the *age* field. A rnoro subtle point is illustrated by the following cOIrllnand, which decrelnents the rating of all 'sailors:

```
UPDATE  Sailors S
SET     S.ratillg = S.rating-l
```

Leah cannot execute this cOlInnand because it requires the SELECT privilege 011 the *S.rating* colurnn anei Leah does not have this privilege.

Bill can refer to the *bid* colurnn of Boats as a foreign key in another table. For example, Bill can create the Reserves table through the following cOlnnland:

```
CREATE TABLE R"eserves (sid    INTEGER,
                        bid    INTEGER,
                        day    DATE,
                        PRIMARY KEY (bid, day),
                        FOREIGN KEY (sid) REFERENCES Sailors ),
                        FOREIGN KEY (bid) REFERENCES Boats)
```

If Bill did not have the REFERENCES privilege on the *bid* coh1111n of Boats, he would not be able to execute this CREATE staternent because the FOREIGN KEY clause requires this privilege. (A sirnilar point holds with respect to the foreign key reference to Sailors.)

Specifying just the INSERT privilege (sirnilarly, REFERENCES and other privileges) in a GRANT conlmand is not the sarne as specifying SELECT(*colurnn-name)* for each column currently in the table. Consider the following command over the Sailors table, which has cohllnns *sid*, *snarne, rating,* and *age:*

```
GRANT INSERT ON Sailors TO J\!Iichael
```

Suppose that this conunand is executed and then a colurnn is added to the Sailors table (by executing an ALTER TABLE cOlIllnand). Note that Michael has the INSERT privilege with respect to the newly added colurnn. If we had executed the following GRANT cornrnand, instead of the previous one, Michael would not have the INSERT privilege on the new cohllnn:

```
GRANT   INSERT ON Sailors(sid), Sailors(sna1ne), Sailors(rating),
        Sailors(age), TO J\!Iichael
```

There is a cornplernentary corl1rnand to GRANT that allows the withdrawal of privileges. The syntax of the REVOKE cOllunand is as follows:

```
REVOKE [GRANT OPTION FOR ] privileges
       ON object FROM users {RESTRICT | CASCADE }
```

The cOIlnnand CH,n be used to revoke either a privilege or just the grant option on a privilege (by using the optional GRANT OPTION FOR clause). One of the two alternatives, RESTRICT or CASCADE, HUlst be specified; we see 'what this choice IneaI1S shortly.

The intuition behind the GRANT command is clear: rrhe creator of a base table or a view is given all the appropriate privileges \vith respect to it and is allowed to pass these privileges—including the right to pass along a privilege—to other users. The REVOKE comuland is, as expected, intended to achieve the reverse: A user who has granted a privilege to another user rnay change his or her lnincI and want to withdraw the gra,nted privilege. The intuition behind exactly 'what effect a REVOKE cornrnand has is conlplicated by the fact that a user Inay be granted the sarne privilege rnultiple tilnes, possibly by different users.

\Vhen a user executes a REVOKE cornmand with the CASCADE keyword, the effect is to \vithdraw the named privileges or grant option froIn all users who currently hold these privileges *solely* through a GRANT cOllunand that was previously executed by the sallIe user who is now executing the REVOKE cOl1nnand. If these users received the privileges with the grant option and passed it along, those recipients in turn lose their privileges as a consequence of the REVOKE cOIurnand, unless they received these privileges through an additional GRANT comIuand.

We illustrate the REVOKE cOllllnand through several examples. First, consider what happens after the following sequence of eornmands, where Joe is the creator of Sailors.

GRANT SELECT ON Sailors TO Art WITH GRANT OPTION *(executed by Joe)*
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION *(executed by Art)*
REVOKE SELECT ON Sailors FROM Art CASCADE *(executed by Joe)*

Art loses the SELECT privilege on Sailors, of course. Then Bob, who received this privilege from Art, and only Art, also loses this privilege. Bob's privilege is said to be **abandoned** when the privilege froIn which it was derived (Art's SELECT privilege with grant option, in this exarnple) is revoked. When the CASCADE keyword is specified, all abandoned privileges are also revoked (possibly causing privileges held by other users to becOlne abandoned and thereby revoked recursively). If the RESTRICT keyword is specified in the REVOKE corll-mand, the cornrnand is rejected if revoking the privileges *just* frorn the users specified in the cOIllluand would result in other privileges becorning abandoned.

Consider the following sequence, as another exarnple:

GRANT SELECT ON Sailors TO Art WITH GRANT OPTION *(executed by Joe)*
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION *(executed by Joe)*
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION *(executed by Art)*
REVOKE SELECT ON Sailors FROM Art CASCADE *(executed by Joe)*

As before, Art loses the SELECT privilege on Sailors. But what about Bob? Bob received this privilege fronl Art, but he also received it independently (coincidentally, directly fro111 Joe). So Bob retains this privilege. Consider a third example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Joe)
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Joe)
REVOKE SELECT ON Sailors FROM Art CASCADE            (executed by Joe)
```

Since Joe granted the privilege to Art twice and only revoked it once, does Art get to keep the privilege? As per the SQL standard, no. Even if Joe absentmindedly granted the saIne privilege to Art several tirnes, he can revoke it with a single REVOKE cOIlunand.

It is possible to revoke just the grant option on a, privilege:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Joe)
REVOKE GRANT OPTION FOR SELECT ON Sailors
        FROM Art CASCADE                             (executed by Joe)
```

This cOlnmand would leave Art with the SELECT privilege on Sailors, but Art no longer has the grant option on this privilege and therefore cannot pass it on to other users.

These exarnples bring out the intuition behind the REVOKE cOillllland, and they highlight the cOlllplex interaction between GRANT and REVOKE cOlnnlands. When a GRANT is executed, a **privilege descriptor** is added to a table of such descriptors Inaintained by the DElVIS. The privilege descriptor specifies the following: the *grantor* of the privilege, the *grantee* who receives the privilege, the *granted privilege* (including the narne of the object involved), and whether the grant option is included. When a user creates a table or view and 'autornatically' gets certain privileges, a privilege descriptor with *system*, as the grantor is entered into this table.

rrhe effect of a series of GRANT cornrnands can be described in terrns of an **authorization graph** in which the nodes are users—technically, they are authorization IDs—and the arcs indicate .how privileges are passed. There is an arc fron1 (the node for) user 1. to user 2 if user 1. executed a GRANT cOIIunand giving a privilege to user 2; the arc is labeled with the descriptor for the GRANT cOIlllnand. A GRANT cOIlnnand has no effect if the saIne privileges ha.ve already been granted to the same grantee by th.e sarne grantor. The following sequence of commands illustrates the sernantics of GRANT and REVOKE connnands when there is a *cycle* in the authorization graph:

```
GRANT SELECT ON Sailors TO .Art WITH GRANT OPTION    (executed by Joe)
GRANT SELECT ON Sa.ilors TO Bob WITH GRANT OPTION    (executed by Art)
```

GRANT SELECT ON Sailors TO Art WITH GRANT OPTION *(executed by Bob)*
GRANT SELECT ON Sailors TO Cal WITH GRANT OPTION *(executed by Joe)*
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION *(executed by Cal)*
REVOKE SELECT ON Sailors FROM Art CASCADE *(executed by Joe)*

The authorization graph for this exarnple is shown in Figure 21.1. Note that we indicate how Joe, the creator of Sailors, acquired the SELECT privilege frorll the DBMS by introdtIcing a *System* node and drawing an arc froIn this node to Joe's node.

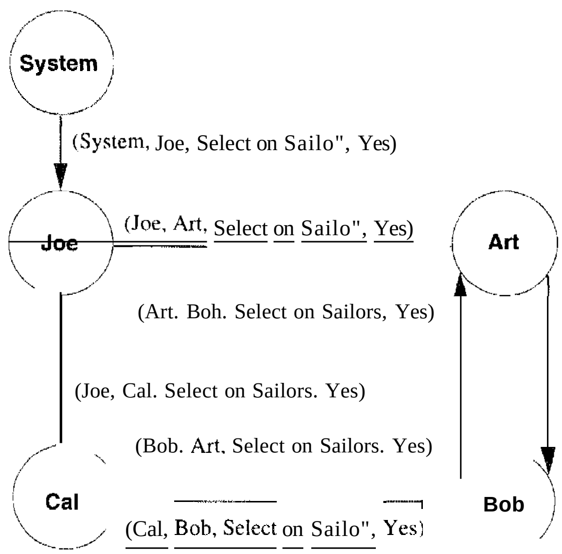


Figure 21.1 Example Authorization Graph

As the graph dearly indicates, Bob's grant to Art and Art's grant to Bob (of the same privilege) creates a cycle. Bob is subsequently given the saIne privilege by Cal, who received it independently froIn Joe. At this point Joe decides to revoke the privilege he granted Art.

Let us trace the effect of this revocation. The arc [raIn Joe to Art is removed because it corresponds to the granting action that is revoked. All rernaining nodes have the following property: *1f node N has an outgoing arc labeled with a privilege, there is a path fTorn the System node to 'node N in 'which each aTC label contains the same privilege plus the grant opt'ion.* That is, any rernaining granting action is justified by a privilege received (directly or indirectly) frorn the Systern. The execution of Joe's REVOKE conllnand therefore stops at this POiIlt,\vith everyone continuing to hold the SELECT privilege on Sailors.

rrhis result may seenl nnintuitive because Art continues to have the privilege only because he received it fr0111 Bob, and at the time that Bob granted the privilege to Art, he had received it only frorn Art. Although Bob acquired the privilege through Cal subsequently, should we not undo the effect of his grant to Art when executing Joe's REVOKE command? 'rhe effect of the grant frorn Bob to Art is *not* undone in SQL. In effect, if a user acquires a privilege rnultiple tilnes frolIl different grantors, SQL treats each of these grants to the user as having occurred *befoTe* that user passed on the privilege to other users. This implementation of REVOKE is convenient in 111any reaJ-\vorld situations. .For exanlple, if a lIl.anager is fired after passing on sorne privileges to subordinates (who lnay in turn have passed the privileges to others), we can ensure that only the rnanager's privileges are rernoved by first redoing all of the Illanager's granting actions and then revoking his or her privileges. That is, we need not recursively redo the subordinates' granting actions.

To return to the saga of Joe and his friends, let us suppose that Joe decides to revoke Cal's SELECT privilege as well. Clearly, the arc frorn Joe to Cal corresponding to the grant of this privilege is rerIloved. The arc frorH Cal to Bob is reilloved as well, since there is no longer a path fronl SystelIl to Cal that gives Cal the right to pass the SELECT privilege on Sailors to Bob. The authorization graph at this interrnediate point is shown in Figure 21.2.

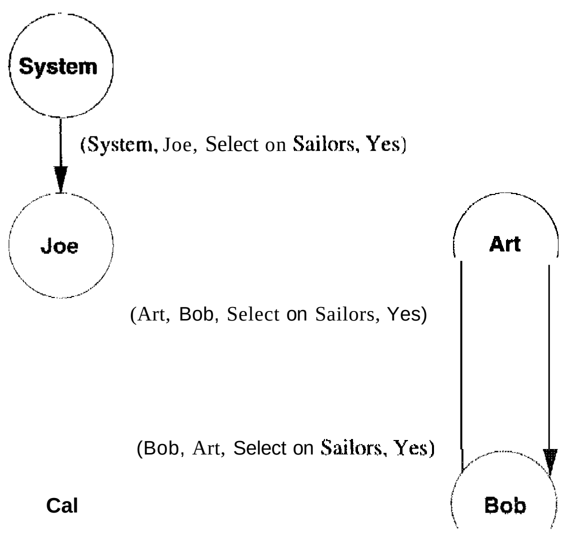


Figure 21.2 Example Authorization Graph during Revocation

rrhe graph now contains two nodes (Art and Bob) for which there are outgoing arcs with labels containing the SELECT privilege on Sailors; therefore, these users have granted this privilege. lInwever, although each node contains an incorning arc carrying the saIne privilege, *there is no such path from Systern to either of these nodes*; so these users' right to grant the privilege has been abandonecL We therefore rernove the outgoing arcs as well. In general, these nodes rnight have other arcs incident on theIn, but in this exarnplc, they now have no incident arcs. Joe is left as the only user\vith the SELECT privilege on Sailors; Art and Bob have lost their privileges.

21.3.1 **Grant** and Revoke on Views and Integrity Constraints

The privileges held by the creator of a view (\vitll respect to the vie\v) change over time as he or she gains or loses privileges on the underlying tables. If the creator loses a privilege held 'with the grant option, users who were given that privilege on the view lose it as \vell. There are SOIHO subtle aspects to the GRANT and REVOKE conunands when they involve views or integrity constraints. We consider senne exarnples that highlight the following irnportant points:

1. A view Inay be dropped because a SELECT privilege is revoked froIn the user who created the view.
2. If the creator of a vie"v gains additional privileges on the underlying tables, he or she autornatically gains additional privileges on the view.
3. The distinction between the REFERENCES and SELECT privileges is irnportanto

Suppose that Joe created Sailors and gave Michael the SELECT privilege on it with the grant option, and J\!Iichael then created the view YoungSailors and gave Eric the SELECT privilege on YoungSailors. Eric now defines a view called FineYoungSailors:

```
CREATE VIEW FineYoungSailors (naIne, age, rating)
       AS SELECT S.snarne, S.age, S.rating
          FROM   YoungSailors S
          WHERE  S.rating > 6
```

What happens if Joe revokes the SELECT privilege on Sailors froln l\1icha,el? Michael no longer has the authority to execute the query used to define YoungSailors because the definition refers to Sailors. rrherefore, the view YoungSailors is dropped (i.e., destroyed). In turn, Fine'{oungSailors is dropped as well. Both view definitions are rernoved fr0111 the systcln catalogs; even if (1, rerIlorseful Joe decides to give ba,ckthe SELECT privilege on Sailors to Michael, the views are gone a11d rnust be created afresh if they are required.

On a Inore happy note, suppose that everything proceeds as just described until Eric defines FineYoungSailors; then, instead of revoking the SELECT privilege on Sailors frorll Michael, .Joe decides to also give Michael the INSERT privilege 011 Sailors. Michael's privileges on the view YoungSailors are upgraded to what he would have if he were to create the vie\v *now*. He therefore acquires the INSERT privilege on 'YourlgSailors as well. (Note that this view is updatal)le.) What about Eric? His privileges are unchanged.

Whether or II.ot Michael has tlle INSERT privilege 011 \roungSailors with the gra11t ()ption depends 011 whether or not .Joe gives hirn the INSERT I)rivilege on

Sailors with the grant option. To understand this situation, consider Eric again. If Michael has the INSERT privilege on YoungSailors with the grant option, he can pass this privilege to Eric. Eric could then insert rows into the Sailors table because inserts on YoungSailors are effected by rnodifying the underlying base table, Sailors. Clearly, we do not want Michael to be able to authorize Eric to rnake such changes unless Michael has the INSERT privilege on Sailors with the grant option.

rrhe REFERENCES privilege is very different froIll the SELECT privilege, as the following exarIlple illustrates. Suppose that Joe is the creator of Boats. He can authorize another user, say, Fred, to create H,eserves with a foreign key that refers to the *bid* colurnn of Boats by giving Fred the REFERENCES privilege with respect to this colulnn. ()n the other hand, if Fred has the SELECT privilege on the *bid* colurnn of Boats but not the REFERENCES privilege, Fred *cannot* create R.eserves with a foreign key that refers to Boats. If Fred creates R,eserves with a foreign key colunlll that refers to *bid* in Boats and later loses the REFERENCES privilege on the *bid* colurnn of boats, the foreign key constraint in Reserves is dropped; however, the R,eserves table is *not* dropped.

To understand why the SQL standard chose to introduce the REFERENCES privilege rather than to siInply allow the SELECT privilege to be used in this situation, consider what happens if the definition of Reserves specified the NO ACTION option with the foreign key------.-Joe, the owner of Boats, Inay be prevented from deleting a row fronl Boats because a row in Reserves refers to this Boats row. Giving Fred, the creator of Reserves, the right to constrain updates on Boats in this rnanner goes beyond. siInply allowing hinl to read the values in Boats, which is all that the SELECT privilege authorizes.

21.4 MANDATORY ACCESS CONTROL

Discretionary access coutrollnechanislns, while generally effective, have certain \veaknesses. In particular they are susceptible to *Trojan horse* schelnes whereby a devious unauthorized user can trick an authorized user into disclosing sensitive data. For exalnple, suppose that student rrricky Dick wants to break into the grade tables of instructor Trustin Justin. IJick does the following:

- He creates a new table called MineAllMine and gives INSERT privileges on this tahle to .Justin (who is blissfully unaware of all this attention, of course).
- He rllodifies the code of SOllle I}BlVIS application that Jllstin uses often to do a couple of additional things: first, read the Grades table, ctlld next, write the result into MineAllMine.

Then he sits back and waits for the grades to be copied into MineAllMine and later undoes the Illodifications to the application to ensure that Justin does not sOlnehow find out later that he has been cheated. Thus, despite the DBMS enforcing all discretionary access controls—only Justin's authorized code was allowed to access Grades—sensitive data is disclosed to an intruder. The fact that Dick could surreptitiously modify Justin's code is outside the scope of the DBMS's access control rnechanisrn.

NIandatory access control meehanisrns are airned at addressing such loopholes in discretionary access control. The popular rllodel for mandatory access control, called the Bell-LaPadula Illodel, is described in tenllS of **objects** (e.g., tables, views, rows, columns), **subjects** (e.g., users, prograrlls), **security classes**, and **clearances**. Each database object is assigned a *security class,* and each subject is assigned *clearance* for a security class; we denote the class of an object or subject *A* as *class(A).* The security classes in a systerll are organized according to a partial order, with a **most secure class** and a **least secure class**. For sirnplicity, we assume that there are four classes: *top secret (T8)*, *secret (8), confidential (C), and unclassified (U).* In this system, *T8> S> C> U,* where *A* > *B* rneans that class *A* data is more sensitive than class *B* data.

The Bell-LaPadula model imposes two restrictions on all reads and writes of database objects:

1. **Simple Security Property:** Subject *S* is allowed to read object 0 only if *class(8)* ≥ *class*(()). For exarllple, a user with *TS* clearance can read a table with C clearance, but a user with C clearance is not allowed to read a table with *TS* classification.

2. ***-Property:** Subject *S* is allowed to write object 0 only if *class(S)* ≤ *class(O).* For exarllple, a user with *S* clearance can write only objects with *S* or *TS* classification.

If discretionary a,ccess controls are also specified, these rules represent additionaJ restrictions. Therefore, to read or write a database object, a user lllUst have the necessary privileges (obtained via GRANT cornrnands) *and* the security classes of the user and the object rnust satisfy the preceding restrictions. Let us consider how such a mandatory control rnech.an.isrn lui.ght h.ave foiled Tricky I)ick. rfhe Grades table could be classified as *S*, .Justin could be given clearance for *S*, and Tricky Di.ck could be given a lower clearance (*C*). Dick can create objects of only *C* or lower classification; so the table MineAllMine can have at Inos1, the classification *C*. When the application prograrIl running on behalf of ..Justin (and therefore\vith clearance *S*) tries to copy Grades into MineAllMine, it is not allowed to do so because *class(MineAllMine)* < *class(applicat'ion),* and the *-Property is violated.

21.4.1 Multilevel Relations and Polyinstantiation

rro apply Inandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of tables, rows, or even individual colurnn values. Let us assU111e that each row is assigned a security class. This situation leads to the concept of a multilevel table, which is a table with the surprising property that users with different security clearances see a different collection of rows when they access the sarne table.

Consider the instance of the Boats table shown in Figure 21.3. Users with *S* and *TS* clearance get both rows in the answer when they ask to see all rows in Boats. A user with C clearance gets only the second row, and a user with *[J* clearance gets no rows.

bid	*bname*	*color*	**Security Class**
101	Salsa	Red	*S*
102	Pinto	Brown	*C*

Figure 21.3 An Instance *B1* of Boats

The Boats table is defined to have *bid* as the prirnary key. Suppose that a user with clearance C wishes to enter the row *(101, Picante, Scarlet, C⟩*. We have a dilemrna:

- If the insertion is perlnitted, two distinct rows in the table have key 101.
- If the insertion is not pennitted because the priInary key constraint is violated, the user trying to insert the new row, who has clearance C, can infer that there is a boat with *bid=101* whose security class is higher than C. This situation cOlnpromises the principle that users should not be able to infer any infonnation about objects that have a higher security classification.

This dilerrllla is resolved by effectively treating the security classification as part of the key. rrhus, the insertion is allo\ved to continue, and the table instance is rnodified as shown in Figure 21.4.

bid	*bna'me*	*color*	**Security Class**
101	Salsa	Red	***S***
101	Picante	Scarlet	***C***
102	Pinto	Brown	***C***

Figure 21.4 Insta.nce 131 after Insertion

lJsers\vith clearance *C* or [1 see just the rows for Picante and Pinto, but users with clearance *S* or *TS* see all three rows. The two ro\vs with *bid*=1()1 can be interpreted in one of two ways: only the rc)\v\vith the higher classification (Salsa, with classification 8) a,ctually exists, or both exist and their presence is revealed to users according to their clearance level. The choice of interpretation is up to application developers and users.

The presence of data objects that appear to have different values to users with different clearances (for exarnple, the boat with *b'id* 101) is called **polyinstantiation**. If we consider security classifications associated with individual colurnns, the intuition underlying polyinstantiation can be generalized in a straightforward manner, but SOlne additional details Inust be addressed. We relnark that the rnain drawback of rnandatory access control schelnes is their rigidity; policies are set by systeIll adrninistrators, and the classification 1nechanisrns are not flexible enough. A satisfactory cornbination of discretionary and rnandatory access controls is yet to be achieved.

21.4.2 Covert Channels, DoD Security Levels

Even if a DElVIS enforces the rnandatory access control schenle just discussed, inforrnation can flow frorn a higher classification level to a lower classification level through indirect rneans, called **covert channels**. For exanlplc, if a transaction accesses data at rnore than one site in a distributed DBMS, the actions at the two sites 1nust be coordina,ted. The process at one site rTlay have a lower clearance (say, C) than the process at another site (say, *S)*, and both processes have to agree to cOllnnit before the transaction can be conunitted. This requirernent can be exploited to pass i11fo1'matio11 with an *S* classification to the process with a *C* clearance: The transaction is repeatedly invoked, and the process \vith the *C* clearance always agrees to cOlIllnit, whereas the process with the *S* clearance agrees to conunit if it wants to translnit a 1 bit and does not agree if it wants to transrnit a 0 l)it.

In this (adrnittedly tortuous) lIlanllcr, infonnation with an *S* clearance can be sent to a process with a *C* clearance as a strea111 of bits. This covert cllannel is an indirect violation of the intent behind the *-Property. Additional exarnples of covert channels can be found readily in statistical databases, which we cliscuss i'n Scetlon 21,.6.2".

DBMS vendors recently started irnplcrnenting rnandatory access control mechaniSlns (although they are not part: of the SQL standard) because the United States l)epartnlent of J)efense (1)01)) requires such support for its systems. The Dol) requirernents can be described in terrns of **security levels** *A*, ,13, *C*, and *D*. of \vhich /1 is the 1J10st secure and 1) is the least secure.

Current Systems: Commercial RDBMSs are available that support discretionary controls at the *C2* level and mandatory controls at the *B1* level. IBM DB2, Inforruix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support SQL's features for discretionary access controL In general, they do not support lnandatory access control; Oracle offers a version of their product with support for rnandatory access control.

Level C requires support for discretionary access control. It is divided into sublevels *Cl* and *C2*; *C2* also requires soiiic degree of accountability through procedures such as login verification and audit trails. Level *B* requires support for lnandatory access control. It is subdivided into levels *Bl, B2,* and *B3.* Level 132 additionally requires the identification and clirnination of covert channels. Level *B3* additionally requires 11laintenance of audit trails and the designation of a security administrator (usually, but not necessarily, the DBA). Level *A,* the most secure level, requires a n1athernatical proof that the security rnechanisrn enforces the security policy!

21.5 SECURITY FOR INTERNET APPLICATIONS

When a DBMS is accessed frorn a secure location, we can rely upon a shnple password rnechanisrn for authenticating users. IIowever, suppose our friend Sarn wants to place an order for a hook over the Internet. rrhis presents sorne unique challenges: Saln is not even a known user (unless he is a repeat custonler). Fron1 Alnazon's point of view, we have an individual asking for a book and offering to pay with a credit card registered to Saln, but is this individual really Sarn? From Sarn's point of view, he sees a fornl asking for credit card inforrnation, but is this indeed a legitirnate part of Arnazon's site, and not a rogue application designed to trick hi1l1 into revealing his credit card nurnber?

Tlhis exarnple illustrates the need for a rnore sophisticated approach to authentication than a sirnple password rnechanisrn. Encryption techniques provide the foun,dation for rnodern authentica,tion.

21.5.1 Encryption

The basic idea behind encryption is to apply an encryption algorithrn to the data, using a user-specified or IJBA-SI)Ccified encryption key. The output of the algorithrn is the encrypted version of the data. There is aJso a decryption algorithrrL ·which takes the encrypted data and a decryption key as input and then returns the original data.\Vithont the correct decryption key, the decryption algoritll111produces gibl)crish. rrhe encryption and clecryption

DES and AES: The DES standard, adopted in 1977, has a 56-bit encryption key. Over time, computers have become so fast that, in 1999, a special-purpose chip and a network of PCs were used to crack DES in under a day. The systern was testing 245 billion keys per second when the correct key was fonnd! It is estimated that a special-purpose hardware device can be built for under a 1l1iUioll dollars that can crack DES in under four hours. Despite growing concerns about its vulnerability, DES is still widely used. In 2000, a successor to DES, called the Advanced Encryption Standard (AES), was adopted as the new (syrrunetric) encryption standard. AES has three possible key sizes: 128, 192, and 256 bits. With a 128 bit key size, there are over $3 . 10^{38}$ possible AES keys, which is on the order of 10^{24} Inore than the number of 56-bit DES keys. Asslllne that we could build a conlputer fast enough to crack DES in 1 second. This COIllputer would. cornpnte for about 149 trillion years to crack a 128-bit AES key. (Experts think the universe is less than 20 billion years old.)

algorithrns thernselves are assunled to be publicly known, but one or both keys are secret (depending upon the encryption scheme).

In symmetric encryption, the encryption key is also used as the decryption key. The ANSI Data Encryption Standard (DES), which has been in use since 1977, is a well-known exarnple of syllunetric encryption. It uses an encryption algorithrn that consists of character substitutions and pernlutations. The nlain weakness of synunetric encryption is that all authorized users rnust be told the key, increasing the likelihood of its becorning known to an intruder (e.g., by sirnple Inllnan error).

Another approach to encryption, called public-key encryption, has becorne increasingly popular in recent years. The encryption scheniC proposed by Hjvest, Sharnir, and Adlernan, called RSA, is a well-known exarnple of public-key encryption. Each authorized user has a public encryption key, known to everyone, and a private decryption key, known only to hini or her. Since the private decryption keys are known only to their owners, the weakness of 1)ES is avoided.

A central issue for public-key encryption is how encryption and decryption keys are chosen. Technically, public-key encryption algorithrns rely on the existence of one-way functions, whose inverses are cornplltationally very hard to deterrnine. rrhe RSA algoritllIn, for example, is based on the observation that, although checking whether a given nurnber is prirne is easy, deterrnining the prirne factors of a nonprime nurnber is extrernely hard. (I)eterlnining the

> Why RSA Works: The essential point of the scherne is that it is easy to compute d given e, p, and q, but *very* hard to cornpute d given just e and L. In turn, this difficulty depends oII the fact that it is hard to determine the prime factors of L, which happen to be p and q. *A caveat:* Factoring is widely believed to be hard, but there is no proof that this is so. Nor is there a proof that factoring is the only way to crack RSA; that is, to comp d frolll e and L.

prirne factors of a nurnber with over 100 digits can take years of CPlJ tirne on the fastest available COIllputers today.)

We now sketch the idea behind the RSA algorithrn, assuming that the data to be encrypted is an integer I. To choose an encryption key and a decryption key for a given user, we first choose a very large integer L, larger than the largest integer we will ever need to encode.[1] We then select a nUl1lber e as the encryption key and cornpute the decryption key d based on e and L; how this is done is central to the approach, as we see shortly. Both Land e are l1lade public and used by the encryption algorithrn. However, d is kept secret and is necessary for decryption.

- The encryption function is S Ie mod L.
- The decryption function is 1 Sd mod L.

We choose L to be the product of two large (e.g., 1024-bit), distinct prirne nurnbers, $p * q$. The encryption key e is a randornly chosen nlunber between 1 and L that is relatively prirne to $(p-1) * (q-1)$. The decryption key d is cornputed such that $d * e = 1 \bmod ((p-1) * (q-1))$. Given these choices, results in nurnber theory can be used to prove that the decryption function recovers the original ruessage frorll its encrypted version.

A very irnportant property of the encryption and decryption algoritluns is that the roles of the encryption and decryption keys can be reversed:

$$decrypt(d, (encrypt(e, I))) = I = decrypt(c, (encrypt(d, I)))$$

Since In.any protocols rely on this property, we henceforth sirnply refer to public and private keys (since both keys can be used for encryption as well as decryption).

[1] A message that is to be encrypted is decomposed into blocks such that each block can be treated as an integer less tha.n L.

While we introduced encryption in the context of authenticatioll, we note that it is a fundaIIlental tool for enforcing security. A DBMS can use *encryption* to protect infornation in situations where the norrnal security rnechanisrns of the DBlVIS are not adequate. For exarnple, an intruder rnay steal tapes containing souic data or tap a conunu.nieation line. By storing and transrnitting data in an encrypted forln, the DBMS ensures that such stolen data is not intelligible to the intruder.

21.5.2 Certifying Servers: The SSL Protocol

Suppose we associate a public key and a decryption key with Alnazon. Anyone, say, user Sam, can send Alnazon an order by encrypting the order using Arnazon's public key. ()nly Arnazon can decrypt this secret order because the decryption algorithrn requires Arnazon's private key, known only to Arnazon.

This hinges on 8arn's ability to reliably find out Arnazon's public key. A number of cornpanies serve as certification authorities, e.g., Verisign. Arnazon generates a public encryption key e_A (and a private decryption key) and sends the public key to Verisign. Verisign then issues a certificate to Arnazon that contains the following infornation:

⟨*Verisign Arnazoin, htl;P8:://www.arnazon. corn,* e_A ⟩

The certificate is encrypted using Verisign's own *private* key, which is known to (i.e., stored in) Internet Explorer, Netscape Navigator, and other browsers.

When 8an1 carnes to the Amazon site and wants to place an order, his browser, running the SSL protocol,[2] asks the server for the Verisign certificate. The browser then validates the certificate by decrypting it (using ·Verisign's public key) and checking that the result is a certificate with the HaIne Verisign, and that theURL it contains is that of the server it is talking to. (Note that an atternpt to forge a certificate will fail because certificates are encrypted using Verisign's private key, whieh is known only to Verisign.) Next, the browser generates a random session key, encrypt it using Arnazon's public key (which it obtained frorn the validated certificate anel therefore trusts), and sends it to the Amazon server.

Frorn this point on, the Arnazon server and the browser can use th.c session key (which both know and are confident tliat only they know) and a *symmetric* encrypticHl protc)collike AES or IJES to exchange securely encrypted rnessages: Messages are encrypted by the sender anel decrypted by the receiver using the sa,HIe session key. rrhe encrypted Inessages travel over the Internet and rnay be

[2] A browser uses the SSL protocol if the target URL begins with *https*.

intercepted, but they cannot be decrypted without the session key. It is useful to consider why \ve need a session key; after all, the bro\vser could sirnply have encrypted 8a1n's original request using Arnazon's public key and sent it securely to the Arnazon server. The reason is that, without the session key, the Amazon server has no way to securely send infonnation back to the bro\vser. A further advantage of session keys is that syrnrnetric encryption is cOlnputationally nluch faster than public key encryption. The session key is discarded at the end of the session.

Thus, 8aIn can be assured that only Alnazon can see the inforrnation he types into the fonn shown to hirn by the AIuazon server and the inforrnation sent back to hiln in responses froIn the server. However, at this point, Amazon has no assurance that the user running the browser is actually Sanl, and not SOlneone who has stolen Sarn's credit card. l-'ypically, rnerchants accept this situation, which also arises when a custoIner places an order over the phone.

If we want to be sure of the user's identity, this can be accoIuplished by additionally requiring the user to login. In our exarnple, 8arn 11lUSt first establish an account with Alnazon and select a password. (Sam's identity is originally established by calling hiln back on the phone to verify the account inforrnation or by sending elnail to an elnail address; in the latter case, all we establish is that the owner of the account is the individual with the given clnail address.) Whenever he visits the site and Anlazon needs to verify his identity, AInazon redirects hinl to a login fo1'1n *after* using SSL to establish a session key. The password typed in is transrnitted securely by encrypting it with the session key.

()ne rcrnaining drawback in this approach is that Arnazon now kno\vs Sarn's credit card nlunber, and he rnust trust Alnazon not to rnisuse it. The Secure Electronic Transaction protocol addresses this lirnitation. Every custolner rnust now obtain a certificate, with his or her own private and public keys, and every transaction involves the Alnazon server, the cust(nner's browser, and the server of a trusted third party, such as Visa for credit card transactions. The basic idea is that the browser encodes non-credit caTd inforrnation using AInazon's public key and the credit ca.rd infonnation using Visa's public key and sends these to the AJnazon server, which for"vards the credit card inforrnation (which it cannot decrypt) to the Visa server. If the Visa server a,pproves the inforrnation, the transa,ction goes through.

21.5.3 Digital Signatures

Suppose tllat ,Elnlcr, who works for ArnazoIl, a,nd Betsy, who works for McGraw-lIill,need to COlllll1Unicate with each other about inventory. Public key encryption can be used t() create digital signatures for rnessages. rrhat is, rnessages can be encoded in such a way that, if Elmer gets a Inessage supposedly fr(nTI Betsy, he can verify that it is fronl Betsy (in addition to being able to decrypt the rnessage) and, further, *prove* that it is froIn Betsy at McGraw-lIill, even if the ll1cssage is sent froIn a IIotrnail account when Betsy is traveling. Sirnilarly, Betsy can authenticate the originator of Inessages froln Ellner.

If Ellner encrypts Inessages for Betsy using her public key, and vice-versa, they can exchange inforrnation securely but cannot authenticate the sender. Sorueone who wishes to irnpersonate Betsy could use her public key to send a rnessage to EIrner, pretending to be Betsy.

A clever use of the encryption sche111e, however, allows Elmer to verify whether the rnessage was indeed sent by Betsy. Betsy encrypts the rnessage using her *private* key and then encrypts the result using Elrner's public key. When Ellner receives such a 111essage, he first decrypts it using his private key and then decrypts the result using Betsy's public key. rrhis step yields the original unencrypted rnessage. Furthermore, Ehner can be certain that the rnessage was composed and encrypted by Betsy because a forger could not have known her private key, and without it the final result would have been nonsensical, rather than a legible 111essage. Further, because even Elmer does not know Betsy's private key, Betsy cannot clairn that Ehner forged the ruessage.

If authenticating the sender is the objective and hiding the rnessage is not important, we can reduce the cost of encryption by using a message signature. A signature is obtained by applying a one-way function (e.g., a hashing schelne) to the rnessagc and is considerably sInaHer. We encode the signature as in the basic digital signature approach, and send the encoded signature together with the full, unencoded 111cssage. rrhe recipient can verify the sender of the signature as just described, and validate the I11essage itself by applying the one-way function and cOlnparing the result with the signature.

21.6 ADDITIONAL ISSUES RELATED TO SECURITY

Security is a l)road topic, and our coverage is necessarily lirnited. 'rhis section briefly touches on sorne additional irnportant issues.

21.6.1 Role of the Database Administrator

rrhe database administrator (IJBA) plays an irnportant role in enforcing the security-related aspects of a database design. In conjunction with the o\vners of the data, the I)BA aJso COlltributes to developing a security policy. The I)BA has a special i:1,ccount, which we call the systenl **account**, and is responsible for the overall security of the systeru. In particular, the DBA. deals with the follo\ving:

1. **Creating New Accounts:** Each new user or group of users Blust be assigned an authorization ID and a password. Note that application prograIns that access the database have the saIne authorization ID as the user executing the prograill.

2. **Mandatory Control Issues:** If the DBMS supports rnandatory control·-···-S0111e custornized systeIns for applications with very high security require-rnents (for exarllple, rnilitary data) provide such support—the DBA lllUst assign security classes to each database object and assign security clear-ances to each authorization ID in accordance with the chosen security pol-ICY·

The DBA is also responsible for rnaintaining the **audit trail**, which is essen-tially the log of updates with the authorization ID (of the user executing the transaction) added to each log entry. This log is just a Ininor extension of the log mechanislll used to recover from crashes. Additionally, the DBA may choose to rnaintain a log of *all* actions, including reads, perfornled by a user. Analyzing such histories of how the DBMS was accessed can help prevent se-curity violations by identifying suspicious patterns before an intruder finally succeeds in breaking in, or it can help track down an intruder after a violation has been detected.

21.6.2 Security in Statistical Databases

A **statistical database** contains specific inforrnation on individuals or events but is intended to perlnit only statistical queries. For exarnple, if we rnailltained a statistical database of inforrna,tion about sailors, we would allow statistical queries about average ratings, rnaxirnurn age, and so on, but not queries about individua.l sailors. Security in such databases poses new probleurs because it is possible to **infer** protected inforrnation (such as a sailor's rating) frorn answers to perrnitted statistical queries. Such inference opportunities represent covert channels that can cornprornise the security policy of the database.

Suppose that sailor Sneaky Pete wants to kncrw the rating of .A.clrniral Ho1'n-tooter, the esteemed chairrnarl of the sailing clu1), and happens to kno\v that IIorntooter is the oldest sailor in the club. Pete repeatedly asks queries of the forln "How InClny sailors are there whose age is greater than X?" for various values of X, until the answer is 1. Obviously, this sa,iIor is lIorntooter, the oldest sailor. Note that each of these queries is a valicl statistical query and is permitted. Let the value of X at this point be, say, 65. Pete no\v asks the query, "What is the nraxirnurn rating of all sailors \vhose age is greater than