65?" Again, this query is pennitted because it is a statistical query. However, the answer to this query reveals J101'ntooter's rating to Pete, and the security policy of the database is violated.

One approach to preventing such violations is to require that each query rnust involve at least S01ne lnininuull nUluber, say, N, of l'()\VS. With a reasonable choice of N, Pete \vould not be able to isolate the inforrnation about 1101'ntooter, because the query about the maximum rating would fail. rrhis restriction, however, is easy to overCOIne. By repeatedly asking queries of the forIII, "How ruany sailors are there whose age is greater than X?" until the systenl rejects one such query, Pete identifies a set *ofN* sailors, including Florntooter. Let the value of X at this point be 55. Now, Pete can ask two queries:

- "What is the SlUll of the ratings of all sailors whose age is greater than 557" Since N sailors have age greater than 55, this query is perrnitted.
- "What is the SUIll of the ratings of all sailors, other than l1orntooter, whose age is greater than 55, and sailor Pete?" Since the set of sailors whose ratings are added up now includes Pete instead of Horntooter, but is otherwise the sallle, the rnunber of sailors involved is still N, and this query is also pennitted.

From the answers to these two queries, say, Al and A_2, Pete, who knows his rating, can easily calculate Horntooter's rating as $Al - A_2 +$ *.Pete'8 rating.*

Pete succeeded because he was able to ask two queries that involved Illany of the sarne sailors. 'The nurnber of rows exalnined in corllrnon by two queries is called their **intersection**. If a limit were to be pla,ced on the alIlount of intersection perrnitted bet\veen any two queries issued by the same user, Pete could be foiled. Actually, a truly fiendish (and patient) user can generally find out inforruation about specific individuals even if the systcrn places a, rniniruurn nUlnber of ro\vs bound (N) and a rnaxirnurn intersection bound (M) on queries, hut the n1.11n])er of queries required to do this gro\vs in proportion to N/M. We can try to additionally lirnit the total nUlnbe1' of queries that a user is allowed to ask. but two users could still conspire to breach security. By Il1aintaining a log of all activity (including read-only accesses), such query patterns can be detected, icleally before a security violation occurs. This discussion should rnake it clear, however, that security in statistical databases is difficult to enforce.

21.7 DESIGN CASE STUDY: THE INTERNET STORE

We return to our case study and our friends at DBI)udes to consider security issues. 'There are three groups of users: custolners, employees, and the owner of the l>ook shop. (()f course, there is also the database adrninistrator, who has universal access to all data and is responsible for regular operation of tlle database systcrTl.)

The owner of the store has full privileges on all tables. Custorners can query the Books table and place orders online, but they should not have access to other customers' records nor to other c11storne1'8' orders.DBDudes restricts access in two ·ways. First, it designs a simple Web page with several fonus similar to the page shown in Figure 7.1 in Chapter 7. This allo\vs custo1ners to subrnit a s111all collection of valid requests without giving tho1n the ability to directly access the underlying DBMS through an SQL interface. Second, I)B.Dudes uses the security features of the DBMS to li111it access to sensitive data.

rIhe \vebpage allows custorners to query the Books relation by ISBN nU111bcr, narne of the author, and title of a book. The webpage also has two buttons. The first button retrieves a list of all of the custolIler's orders that are not completely fulfilled yet. l'he second button displays a list of all cornpleted orders for that custorner. Note that custolllers cannot specify actual SQL queries through the Web but only fill in SCHne pararneters in a forn1 to instantiate an autonlatically generated SQL query. All queries generated through fonll input have a WHERE clause that includes the *cid* attribute value of the current custo1ner, and evaluation of the queries generated by the two buttons requires knowledge of the custolller identification nUlnber. Since all users have to log on to the website before browsing the catalog, the business logic (discussed in Section 7.7) lllust 1naintain state inforrnation about a custoDler (i.e., the Cllstorner identification nUlnber) during the custorner's visit to the website.

The second step is to configure the database to lirnit access according to each user group's need to know. DBI)udes creates a special customer account that has the following privileges:

```
SELECT ON Books, NewOrders, ()ldOrders, NewOrderlists, OldOrderlists
INSERT ON New()rders, OldOrders, New()rderlists, ()ld()rderlists
```

Ernployees should be able to acid new books to the catalog, upda,te the quantity of a book in stock, revise custorner orders if necessary, and update all custorner inforrnation *except the credit card information.* In fact, ernployees should not even be able to see a custorner's credit card nurnber. 1]lcreforc,DBDucles creates the following view:

```
CREATE VIEW CustomerInfo (cid,cnarnc,address)
       AS SELECT C.cid, C.cname, C.(l.cldress
          FROMCllstolners C
```

I)BI)udes gives the employee account the follc)\ving privileges:

```
SELECT ON CustomerInfo, Books,
          NewOrders, ()IdOrders, NewOrderlists, Old()rderlists
INSERT ON Cllstornerlnfo, Books,
          Nc\vOrders, 01dC)rders, NewOrderlists, ()ldOrderlists
UPDATE ON CustolnerInfo, Books,
          New()rders, OldOrders, NewOrderlists, Old()rderlists
DELETE ON Books, NewOrders, OldOrders, NewOrderlists, ()ldOrderlists
```

Observe that ernployees can rnodify Custornerlnfo and even insert tuples into it. This is possible because they have the necessary privileges, and further, the view is updatable and insertable-into. While it seerns reasonable that elllployees can update a custorner's address, it does sceln odd that they can insert a t1lple into Cllstornerlnfo even though they cannot see related infonna.tion about the custorner (i.e., credit card nurnber) in the Cllstorners tahle. The reason for this is that the store wants to be able to take orders 1'1'0111 first-tirne custorners over the phone without asking for credit card inforrnation over the phone. Ernployees can insert into Custornerlnfo, effectively creating a new Custoillers record without credit card inforluation, and custorners can subsequently provide the credit card nurnber through a Web interface. (Obviously, the order is not shipped until they do this.)

In addition, there are security issues when the user first logs on to the website using the cllstolner identification nUlnber. Sending the nUlnber unencrypted over the Internet is a security hazard, and a secure protocol such as SSL should be used.

Cornpanies such as CyberCash and DigiCash offer electronic conunerce pay-rIlcnt solutions, even inclu.ding *electronic cash.* Discussion of how to incorporate such techniques into the website are outside the scope of this book.

21.8 REVIEW QUESTIONS

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

- What are tIle 1n,ain objectives in designing a secure datal)ase application? Explain the te1'ms *secrecy*, *integrity*, *availability*, and *authentication.* (Section 21.1)
- Explain the terms *security policy* and *security mechanism* arid how tllCy are related. (Section 21.1)
- What is the Blain idea behind *discretionary access control*? What is the idea behind *mandatory access control*? What are the relative merits of these two approaches? (Section 21.2)

- Describe the privileges recognized in SQL? In particular, describe SELECT, INSERT, UPDATE, DELETE, and REFERENCES. For each privilege, indicate \vho acquires it automatically on a given table. (Section **21.3)**
- I-Io\v are the owners of privileges identified? In particular, discuss *authorization ID8* and *roles.* (Section 21.3)
- What is an *authorization graph*? Explain SQL's GRANT and REVOKE eO111-mands in terrns of their effect on this graph. In particular, discuss what happens when users pass on privileges that they receive frorn sorneone else. (Section **21.3)**
- Discuss the difference between having a privilege on a table and on a vie\v defined over the table. In particular, how can a user have a privilege (say, SELECT) over a view 'without also having it on all underlying tables? Who 11111st have appropriate privileges on all underlying tables of the view? (Section 21.3.1)
- What are *objects, subjects, security classes,* and *cleaTances* in rnandatory access control? IJiscuss the Bell-LaPadula restrictions in tenns of these concepts. Specifically, define the *simple security property* and the **-pToperty.* (Section 21.4)
- What is a *Trojan horse* attack and how can it cOlnprornise discretionary access control? Explain how lnandatory a,ccess control protects against Trojan horse attacks. (Section 21.4)
- What do the tenns *multilevel table* and *polyinstantiation* mean? Explain their rela.tionship, a,nd how they arise in the context of lnandatory access control. (Section 21.4.1)
- What are *covert channels* and how can they arise when both discretionary and luandatory access controls are in place? (Section **21.4.2)**
- Discuss the I)oD security levels for database systclns. (Section 21.4.2)
- Explain why a sirnple password rnechanisrn is insufficient for authentication of users who access a database renJotely, say, over the Internet. (Section 21.5)
- What is the difference between *symmetric* and *public-key encryption*? Give examples of well-known encryption algoritluns of both ki.lldso What is the rnain weakness of synunetric encryption and how is this addressed in public-key encryption? (Section 21.5.1)
- 1)i8c118s the choice of encrYIltion and decryption keys in public-key en.cryption and how they are llsed to encrypt and decrypt data. Explain the role of *one-way functions.* What H.ssurance do\ve h.ave that the RSA scheme cannot be cornprornised? (Section 21.5.1)

- What are *certification authorities* and why are they needed? Explain how *certificates* are issued to sites and validated by a bro\vser using the *SSL protocol*; discuss the role of the *session key*. (Section 21.5.2)
- If a user connects to a site using the SSL protocol, explain \vhy there is still a need to login the uscr. Explain the use of SSL to protect pass\vords and other sensitive infonnation being exchal1ged. What is the *secure electronic transaction protocol*? What is the added value over SSL? (Section 21.5.2)
- A *digital signature* facilitates secure exchange of rnessages. Explain what it is and how it goes beyond sirnply encrypting rnessages. Discuss the use of *message signatures* to reduce the cost of encryption. (Section 21.5.3)
- What is the role of the database achninistrator with respect to security? (Section 21.6.1)
- Discuss the additional security loopholes introduced in *statistical databases.* (Section 21.6.2)

EXERCISES

Exercise 21.1 Briefly answer the following questions:

1. Explain the intuition behind the two rules in the Bell-LaPadulamodel for rnandatory access control.
2. Give an exarnple of how covert channels can be used to defeat the Bell-LaPadula rnodel.
3. Give an exarnple of polyinstantiation.
4. Describe a scenario in whichrnandatory access controls prevent a breach of security that cannot be prevented through discretionary controls.
5. Describe a scenario in which discretionary access controls are required to enforce a seCll-rity policy that cannot be enforced using only mandatory controls.
6. If a DBMS already supports discretionary and Jnandatory access controls, is there a need for encryption?
7. Explain the need for each of the following lirnits in a statistical database systern:
   (a) A maxirnurn on the munber of queries a user can pose.
   (b) A rninirnUln on the munber of tuples involved in ans\vering a query.
   (c) A maximurn on the intersection of two queries (i.e., on the number of tuples that both queries exarnine).
8. Explain the use of an audit trail, with special reference to a statistical database system.
9. \,Vllat is the role of the DBA with respect to security?
10. Describe AES and its relationship to DES.
11. What is public-key encryption? How does it differ frorn the encryption approach taken in the Data Encryption Standard (DES), and in what ways is it better than DES?

12. Explain how a company offering services on the Internet could use encryption-based techniques to lllake its order-entry process secure. Discuss the role of DES, A.ES, SSL, SET, and digital signatures. Search the Web to find out IllOre about related techniques such as *electronic cash*.

Exercise 21.2 You are the DBA for the VeryFine Toy Cornpany and create a relation called Employees with fields *enam,e, dept,* and *salary.* For authorization reasons, you also define views EmployeeNallIes (with *ena:rne* as the only attribute) and DeptInfo with fields *dept* and *avgsalary.* The latter lists the average salary for each departrnent.

1. Show the view definition statements for EnlployeeNames and Deptlnfo.
2. What privileges should be granted to a user who needs to know only average departn1ent salaries for the Toy and CS departments?
3. You want to authorize your secretary to fire people (you will probably tell hill1 whorn to fire, but you want to be able to delegate this task), to check on who is an elllployee, and to check on average department salaries. What privileges should you grant?
4. Continuing with the preceding scenario, you do not want your secretary to be able to look at the salaries of individuals. Does your answer to the previous question ensure this? Be specific: Can your secretary possibly find out salaries of *some* individuals (depending on the actual set of tuples), or can your secretary always find out the salary of any individual he wants to?
5. You want to give your secretary the authority to allow other people to read the EUlployeeNames view. Show the appropriate conI111and.
6. Your secretary defines two new views using the EnIployeeNarnes view. The first is called AtoRNames and simply selects names that begin with a letter in the range A to R. The second is called HowManyNan1es and counts the number of narnes. You are so pleased with this achievement that you decide to give your secretary the right to insert tuples into the EnlployeeNan1es view. Show the appropriate cOllunand and describe 'what privileges your secretary has after this cornrnand is executed.
7. Your secretary allows Todd to read the Erl1ployeeNarnes relation and later quits. You then revoke the secretary's privileges. \Vhat happens to Todd's privileges?
8. Give an exarnple of a view update on the preceding schelna that cannot be illlplernentecl through updates to Erl1ployees.
9. You decide to go on an extended vacation, and to rnake sure that ernergencies can be handled, you want to authorize your boss Joe to read and modify the Employees relation and the ErIlployeeNalnes relation (and Joe lllust be able to delegate authority, of course, since he is too far up the managernent hierarchy to actually do any \vork). Show the appropriate SQL staternents. Can Joe read the Deptlnfo view?
10. After returning frorn your (wonderful) vacation, you see a note from Joe, indicating that he authorized his secretary Mike to read the Ernployees relation. You \vant to revoke Mike's SELECT privilege on Ernployees, but you do not \vant to revoke the rights you gave to Joe, even teruporarily. Can you do this in SQL?
11. Later you realize that Joe has been quite busy. He has defined a view called AllNarnes using the view ErnployeeNames, defined another relation called StaffNarnes that he has access to (but you cannot access), and given his secretary Mike the right to read from the AllNames view. Mike has passed this right on to his friend Susan. You decide that, even at the cost of annoying Joe 1)y revoking Borne of his privileges, you sirnply have to take away Mike (\,nd Susarl's rights to see your data. What REVOKE staternent \vould you execute? What rights does Joe have on Ernployees after this statement is executed? What views are dropped as a consequence?

22

PARALLEL AND DISTRIBUTED DATABASES

- What is the rnotivation for parallel and distributed DBMSs?
- What are the alternative architectures for parallel database systellls?
- How are pipelining and data partitioning used to gain parallelism?
- How are dataflow concepts used to parallelize existing sequential code?
- What are alternative architectures for distributed DBMSs?
- How is data distributed across sites?
- How can we evaluate and optimize queries over distributed data?
- What are the nlerits of synchronous vs. asynchronous replication?
- How are transactions Inanaged in a distributed environment?
- Key concepts: parallel DBMS architectures; perfonnance, speed-up and scale-up; pipelined versus data-partitioned parallelism, blocking; partitioning strategies; dataflow operators; distributed DBMS architectures; heterogeneous systernsj gateway protocols; data distribution, distributed catalogs; sernijoins, data shipping; synchronous versus asynchronous replication; distributed transactions, lock nlanagcrnent, deadlock detection, two-phase ccnnlnit, Presurned Abort

No rnan IS an island, entire of itself; every Tnan IS a pIece of the contirlcnt, a part of the rnain.

—John Donne

In this chapter we look at the issues of parallelism and data distribution in a DBMS. We begin by introducing parallel and distributed database systcrIls in Section 22.1. In Section 22.2, we discuss alternative hardwa,re configurations for a parallel DBMS. In Section 22.3, we introduce the concept of data partitioning and consider its influence on parallel query evaluation. In Section 22.4, we show how data partitioning can be used to parallelize several relational operations. In Section 22.5, we conclude our treatrnent of parallel query processing with a discussion of parallel query optirnization.

'The rest of the chapter is devoted to distributed databases. We present an overview of distributed databases in Section 22.6. We discuss sorne alternative architectures for a distributed DBMS in Section 22.7 and describe options for distributing data in Section 22.8. We describe distributed catalog rnanagernent in Section 22.9, then in Section 22.10, we discuss query optirnization and evaluation for distributed databases. In Section 22.11, we discuss updating distributed data, and fina.lly, in Sections 22.12 to 22.14 we describe distributed transaction ruanagernent.

22.1 IN"TRODUCTION

We have thus far considered centralized database rnanageruent systerns in which all the data is luaintained at a single site and assumed that the processing of individual transactions is essentially sequential. One of the most irnportant trends in data.bases is the increased use of parallel evaluation techniques and data, distribution.

A **parallel database system** seeks to irnprove performance through parallelization of various operations, such as loading data, building indexes, and evaluating queries. Although data may be stored in a distributed fashion in such a systcrn, the distribution is governed. solely by perfon.nance considerations.

In a **distributed database systenl**, data, is physically stored across several sites, and each site is typically rnanaged by a DBMS capable of running il1-dependent of the 01:11e1' sites. rrhe location of data itenlS and the degree of autonorny of iJldividual sites have a significant irnpa,ct on all aspects of the system, including query optirnization and processing, concurrency control, and recovery. In contrast to parallel databases, the distribution of data is governed by factors such as locaJ ownership and increased a,vailability, in addition to perforlnance issues.

While parallelism is 1110tivated l)y performance consideratiolls, several distinct issues rnotivate data distribution:

- Increased AvailabHity: If a site containing a relation goes down, the relation continues to be available if a copy is Inaintained at another site.
- Distributed Access to Data: An organization Inay have branches in. several cities. Although analysts may need to access data corresponding to different sites, we usually find locality in the access pa,tterns (e.g., a bank lnanager is likely to look up the accounts of custorners at the local branch), and this locality can be exploited by distributing the data accordingly.
- Analysis of Distributed Data: Organizations \vant to examine all the data available to thern, even when it is stored across rnultiple sites and on lllultiple database systerns. Support for such integrated access involves nlany issues; even enabling access to widely distributed data can be a challenge.

22.2 ARCHITEC"rURES FOR PARALLEL DATABASES

The basic idea behind parallel databases is to carry out evaluation steps in parallel whenever possible, and there are rnany such opportunities in a relational DBMS; databases represent one of the lnost successful instances of parallel cornputing.

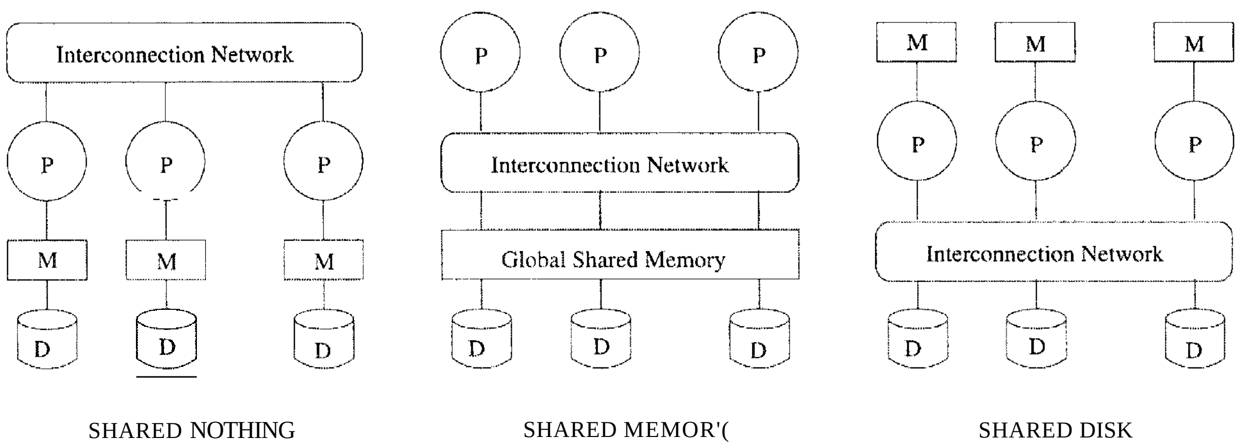


Figure 22.1 PhysicaJ Architectures for Parallel Da.tabase Systems

Three luain architectures have been proposed for building parallel DBIVISs. In a shared-IuerIlory SystCIll, Inultiple CPU·s are attached to an interconnection net\vork and can access a cornrl1on region of rnain lnelilory. In a. shared-disk s:ysten.1, each CPU has a private rnelnory and direct access to all disks through an interconnection network. In a shared-nothing system, each CPTJ has local rnain lnelnory and disk space, but no two CP1Js can access the sarne storage area; all cOHununication between CP1Js is tllrough a 11etwork connection.. rrhe three architectures are illustrated in Figure 22.1.

The shared-rnernory architecture is closer to a conventional machine, and Illany conunercial database systerns have been ported to shared lnernory platfornlS \vith relative ease. Communication overhead is low, because lnain rncIllory can be used for this purpose, and operating systern services can be leveraged to utilize the additional CPUs. Although this approach is attractive for achieving rnoderate parallelism—a few tens of CPlJs can be exploited in this fashion- Inernory contention bec01nes a bottleneck as the nurnber of CPUs increases. rfhe shared-disk architecture faces a sirnilar problcrn because large a1110nnts of data are shipped through the interconnection network.

The basic problern with the shared-111Crrlory and shared-disk architectures is interference: As Inore CPUs are added, existing CPUs are slowed down because of the increased contention for mClllory accesses and network bandwidth. It has been noted that even an average 1 percent slowdown per additional CPU 1nea11S that the rnaxirnum speed-up is a factor of 37, and adding additional CPlJs actually slows down the systern; a systenl with 1000 CPUs is only 4 percent as effective as a *single-CPU* systern! This observation has rllotivated the developrnent of the shared-nothing architecture, which is now widely considered to be the best architecture for large parallel database systems.

rrhe shared-nothing architecture requires rnore extensive reorganization of the DBNIS code, but it has been shown to provide linear speed-up, in that the tilne taken for operations decreases in proportion to the increase in the nUlnber of CPlJs and disks, and linear scale-up, in that perforrnance is sustained if the nurnber of CPUs and disks are increased in proportion to the arnount of data. Consequently, ever-rnore-powerful parallel database systcrns can be built by taking advantage of rapidly irnproving perforrllance for single-CPU systelns and connecting as rnany CPUs as desired.

Speed-up and scale-up are illustrated in Figure 22.2. 'The speed-up curves show how, for a fixed database size, Inore transactions can be executed l)cr second by adding CPUs. The scale-up curves show how adding Inorc resources (in the forln of CPlJs) enables us to process larger problerns. rrhe first scale-up graph Incasures the nurnber of transactions executed per second as the clatabase size is increased and the nurnber of CPlJs is correspondingly increased. Arl alternative way to Ineasure scale-up is to consider the time taken per transaction as r1101'e CPUs are added to process an increasing nurnber of transactions per second; the goal here is to sustain the response tirne per transaction.

22.3 PARALLEL QUERY EVALUATION'

In this section, we discuss parallel evaluation of a relational query in a DBMS with a shared-nothing architecture. While it is possible to consicler parallel

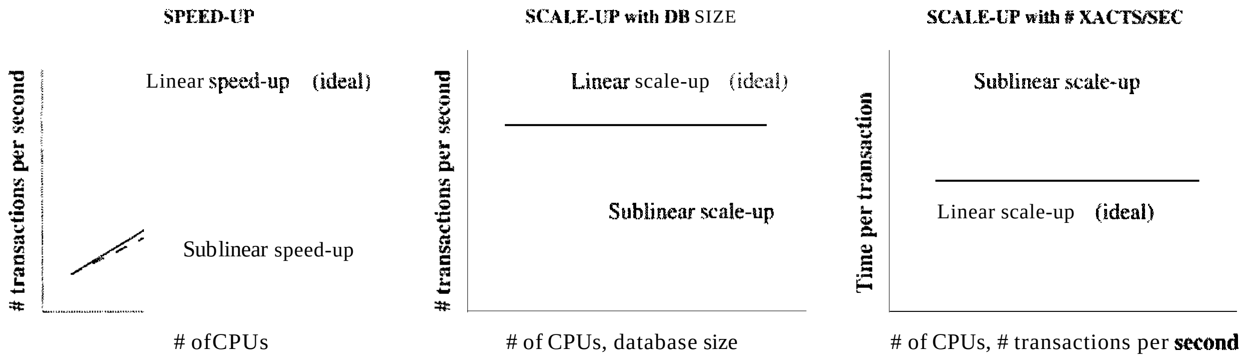


Figure 22.2 Speed-up and Scale-up

execution of rnultiple queries, it is hard to identify in advance which queries will run concurrently. So the ernphasis has been on parallel execution of a single query.

A relational query execution plan is a graph of relational algebra operators, and the operators in a graph can be executed in parallel. If one operator consurnes the output of a second operator, we have **pipelined parallelism** (the output of the second operator is worked on by the first operator as soon as it is generated); if not, the two operators can proceed esseptially independently. An operator is said to **block** if it produces no output until it has conSUllled all its inputs. Pipelined parallelisrn is lirnited by the presence of operators (e.g., sorting or aggregation) that block.

In addition to evaluating different operators in parallel, we can evaluate each individual operator in a query plan in a parallel fashion. rrhe key to evaluating an operator in pa,rallel is to *partition* the input data; \ve can then work on each partition in parallel and cornbine the results. This approach is called **data-partitioned parallel evaluation**. By exercising sorne care, existing code for sequentially evaluating relational operators can be ported easily for data-partitioned parallel evaluation.

An inlportant observation, which explains why shared-nothing parallel database systelns have been very successful, is that database query evaluation is very amenable to data-partitioned parallel evaluation. The goal is to nlinirnize data shipping b:y paTtitioning the data and structuring the algoritluns to do lnost of the processing at individual processors. (We lIse *processor* to refer to a CPU together with its local disk.)

We now consider data paxtitioning and parallelization of existing operator evaluation code in rnore detail.

22.3.1 Data Partitioning

Partitioning a large dataset horizontally across several disks enables us to exploit the I/O banchvidth of the disks by reading and writing theln in parallel. rrhere are several ways to horizontally partition a relation.vVe can assign tuples to processors in a round-robin fashion, we can use hashing, or we can assign tuples to processors by ranges of field values. If there are n processors, the 'ith tuple is assigned to processor *i rnodn* in **round-robin partitioning**. Recall that round-robin partitioning is used in RAID storage systelTIS (see Section 9.2). In **hash partitioning**, a hash function is applied to (selected fields of) a tuple to deternline its processor. In **range partitioning**, tuples are sorted (conceptually), and n ranges are chosen for the sort key values so that each range contains roughly the SalTIe nurnber of tuples; tuples in range i are assigned to processor i.

Round-robin partitioning is suitable for efficiently evaluating queries that access the entire relation. If only a subset of the tuples (e.g., those that satisfy the selection condition *age* = 20) is required, hash partitioning and range partitioning are better than round-robin partitioning because they enable us to access only those disks that contain rnatching tuples. (Of course, this statement assumes that the tuples are partitioned on the attributes in the selection condition; if *age* = 20 is specified, the tuples must be partitioned on *age.)* If range selections such as 15 < *age* < 25 are specified, range partitioning is superic)!' to hash partitioning because qualifying tuples are likely to be clustered together on a few processors. On the other hand, range partitioning can lead to **data skew**; that is, partitions with widely varying numbers of tuples across partitions or disks. Skew causes processors dealing with large partitions to becorne perfonnance bottlenecks. Hash partitioning has the additional virtue that it keeps data evenly distributed even if the data grows and shrinks over tirne.

To reduce skew in range partitioning, the luain question is how to choose the ranges by which tuples are distributed. ()ne effective approach is to take sarnpIes fronl each processor, collect and sort all sarnples, and divide the sorted set of samples into equally sized subsets. If tuples are to be partitioned on *age*, the *age* ranges of the sarnpled subsets of tuples can be used as the basis for redistributing the entire relation.

22.3.2 Parallelizing Sequential Operator Evaluation Code

An elegant software architectnre for parallel DBMSs enables us to readily parallelize existing code for sequentially evaluating a relational oI>crator. The basic idea is to use parallel da.ta strearrlS. Streams (frorn different disks or the output of other operators) are Inerged as needed to provide the inputs for a relational operator, and the output of an operator is split as needed to parallelize subsequent processing.

A. parallel evaluation plan consists of a dataflow network of relational, luerge, and split operators. 1'he rnerge and split operators should be able to buffer sOlne data and should be able to halt the operators producing their input data. They can then regulate the speed of the execution according to the execution speed of the operator that conSUlues their output.

As we will see, obtaining good parallel versions of algorithllls for sequential operator evaluation requires careful consideration; there is no luagic formula for taking sequential code and producing a parallel version. Good use of split and 111erge in a dataflow software architecture, however, can greatly reduce the effort of implementing parallel query evaluation algorithms, as we illustrate in Section 22.4.3.

22.4 PARALLELIZING INDIVIDUAL OPERATIONS

This section shows how various operations can be implemented in parallel in a shared-nothing architecture. We assurne that each relation is horizontally partitioned across several disks, although this partitioning mayor may not be appropriate for a given query. The evaluation of a query must take the initial partitioning criteria into account and repartition if necessary.

22.4.1 Bulk Loading and Scanning

We begin with two simple operations: *scanning* a relation and *loading* a relation. Pages can be read in parallel while scanning a relation, and the retrieved tuples can then be lnerged, if the relation is partitioned across several disks. More generally, the idea also applies when retrieving all tuples that Incet a selection condition. If hashing or range partitioning is used, selection queries can be answered by going to just those processors that contain relevant tuples.

A sirnilar observation holds for bulk loading. Further, if a relation has associated indexes, any sorting of data entries required for building the indexes during bulk loading can also be done in parallel (see later).

22.4.2 Sorting

A sirnple idea is to let each CPTJ sort the part of the relation that is on its local disk and then rnerge these sorted sets of tuples. The degree of parallelisHl is likely to be lirnited by the rnerging phase.

A better idea is to first redistribute all tuples in the relation using range partitioning. For exarnple, if we want to sort a collection of employee tuples by salary, salary values range froIH 10 to 210, and we have 20 processors, we could send all tuples with salary values in the range 10 to 20 to the first processor, all in the range 21 to 30 to the second processor, and so on. (Prior to the redistribution, while tuples are distributed across the processors, we cannot assurne that they are distributed according to sa1ary ranges.)

Each processor then sorts the tuples assigned to it, using sorne sequential sorting algorithrn. For exaluple, a processor can collect tuples until its lllemory is full, then sort these tuples and write out a run, until all incolning tuples have been written to such sorted runs on the local disk. rrhese runs can then be rnerged to create the sorted version of the set of tuples assigned to this processor. The entire sorted relation can be retrieved by visiting the processors in an order corresponding to the ranges assigned to thenl and sirnply scanning the tuples.

The basic challenge in parallel sorting is to do the range partitioning so that each processor receives roughly the same runnber of tuples; otherwise, a processor that receives a disproportionately large nurnber of tuples to sort becornes a bottleneck and lirnits the scalability of the parallel sort. ()ne good approach to range partitioning is to obtain a sarnple of the entire relation by taking sarnples at each processor that initially contains part of the relation. The (relatively srnall) saruple is sorted and used to identify ranges with equal nUlllbers of tuples. This set of range values, called a **splitting vector**, is then distributed to all processors and used to range partition the entire relation.

A particularly irnportant application of parallel sorting is sorting the data entries in tree-structured indexes. Sorting data entries can significantly speed up the process of bulk-loading an index.

22.4.3 Joins

In this section, we consider how the join operation can be parallelized.\Ve present the basic idea behind the parallelization and illustrate the use of the rnerge and split operators described in Section 22.:3.2. We focus on parallel hash join, which is widely used, and briefly outline how sort-rnerge join ca,n be similarly parallelized. ()ther join algorithniS can be parallelized as well, although not as effectively as these two algoritlnns.

Suppose that we want to join two relations, say, A and B, on the *age* attribute. We aSSUIllC that they are initially distribl.lted across several disks in senne way that is not useful for the join operation; that is, the initial partitioning is not based on the join attribute. The basic idea for joining A and B in parallel is to decornpose the join into a collection of k srnnller joins. We can decornpose the join by partitioning both A and B into a collection of k logical buckets or partitions. By using the sarne partitioning function for both A and B, we ensure that the union of the k s1naller joins cOlnputes the join of A and B; this idea is si1nilar to intuition behind the partitioning phase of a sequential hash join, described in Section 14.4.3. Because A and B are initially distributed across several processors, the pa,rtitioning step itself can be done in parallel at these processors. At each processor, all local tuples are retrieved and hashed into one of k partitions, with the salIie hash function used at all sites, of course.

Alternatively, we can partition A and B by dividing the range of the join attribute *age* into k disjoint subranges and placing A and B tuples into partitions according to the subrange to which their *age* values belong. For exanlple, suppose that \ve have 10 processors, the join attribute is *age*, with values froln 0 to IOO. Assurlling uniforrl1 distribution, A and B tuples with $0 \leq age < 10$ go to processor 1, $10 \leq age < 20$ go to processor 2, and so on. This approach is likely to be 1110re susceptible than hash partitioning to data skew (i.e., the number of tuples to be joined can vary widely across partitions), unless the subranges are carefully deterrnined; we do not discuss how good subrange boundaries can be identified.

I-Iaving decided on a partitioning strategy, we can assign each partition to a processor and carry out a local join, using any join algorithrl1 we want, at each processor. In this case, the nUIlber of partitions k is chosen to be equal to the nUlnber of processors n available for carrying out the join, and during partitioning, each processor sends tuples in the ith partition to processor i. After partitioning, each processor joins the A andB tuples assigned to it. Each join process executes sequential join code and receives input .4. and 13 tuples froro several processors; a rnerge operator lnerges all incorning A tuples, and another merge operator rnerges all incorning B tuples. Depending on 110\v we want tC) distribute the result of the join of A and B, the output of the join process rl1ay be split into several data strealIlS. The network of operators for parallel join is sho\vn in Figure 22.3. To sirnplify the figure, we assurlle that the proc.essors doing the join are distinct frorn the processors that. initially contain tuples of A and B and show only four processors.

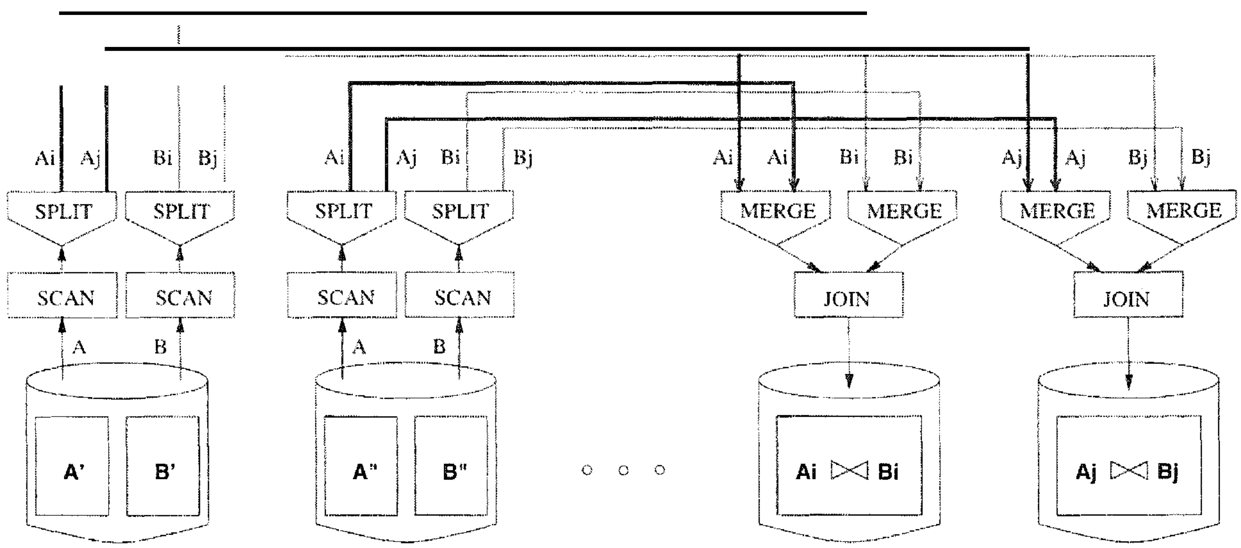


Figure 22.3 Dataflow Network of Operators for Parallel Join

If range partitioning is used, this algorithrn leads to a parallel version of a sort-merge join, with the advantage that the output is available in sorted order. If hash partitioning is used, we obtain a parallel version of a hash join.

Improved Parallel Hash Join

A hash-based refinernent of the approach offers improved perforruanee. The ruain observation is that, if A and B are very large and the nurnber of partitions k is chosen to be equal to the nurnber of processors n, the size of each partition 111ay still be large, leading to a high cost for each local join at the n processors.

An alternative is to execute the srnaller joins $A_i \bowtie B_i$, for $i = 1 \ldots k$, one after the other, but\vith each join executed in parallel using all processors. This approa,ch allows us to utilize the total available ruain rueruory at all n processors in each join $A_i \bowtie 13_i$ and is described in rnore detail as follcJ\vs:

1. At each site, apply a hash function hI to partition the A and B tuples at this site into partitions $i = 1 \ldots k$. Let A be the srnaller relation. The nurnber of partitions k is chosen such that each partition of A fits into the *aggregate* or cornbined rnernory of all n processors.

2. For $i = 1 \ldots k$, process the join of the ith partitions of A and B. To cornpute $A_i \bowtie B_i$, do the follcnving at every site:
   - (a.) i\pply a second hash function 12,2 to all A_i tuples .to detennine where they should be joined and send tuple t to site $h2(t)$.
   - (b) As A_i tuples arrive to be joined, add thcln to an in-rnernory hash. table.

(c) After all A_i tuples have been distributed, apply *h2* to B_i tuples to deterrnine where they should be joined and send tuple *t* to site *h2(t)*.

(d) As B_i tuples HJTive to be joined, probe the in-rnernory table of A_i tuples and output result tuples.

The lIse of the second hash function *h2* ensures that tuples are (rIlore or less) uniforrnly distributed across all *n* processors participating in the join. This approach greatly reduces the cost for each of the srnaller joins and therefore reduces the overall join cost. ()bserve that all available processors are fully utilized, even though the srnaller joins are carried out one after the other.

The reader is invited to adapt the network of operators shown in Figure 22.3 to reflect the improved parallel join algorithrn.

22.5 PARALLEL QUERY OPTIMIZATION

In addition to pa.rallelizing individual operations, we can obviously execute different operations in a query in parallel and execute rnultiple queries in parallel. Optirnizing a single query for parallel execution has received rnore attention; systerus typically optirnize queries without regard to other queries that might be executing at the same tilne.

rrwo kinds of interoperatioll parallelisrn can be exploited within a query:

- The result of one operator can be pipelined into another. For example, consider a left-deep plan in which all the joins use index nested loops. The result of the first (i.e., the bottollunost) join is the outer relation tuples for the next join node. As tuples are produced by the first join, they can be used to probe the inner relation in the second join. T'he result of the second join can sirnilarly be pipelined into the next join, and so 011.
- Multiple independent operations can be executed concurrently. For exarnpIe, consider a (bushy) plan in vilhich relations *A* and *B* are joined, relations *C* and *D* are joined, and the results of these two joins are finally joined. Clearly, the join of *A* and *B* can be executed conculTcntly with the join of *C* and *D*.

A.n optirnizer that seeks to parallelize query evaluation has to consider several issues, and we only outline the rnain points. The cost of executing individual operations in paraJlel (e.g., parallel sorting) obviollsly differs frorn executing thern sequentially, and the optirnizer should estimate operation costs accordingly.

Next, the plan that returns answers quickest Inay not be the plan with the least cost. For example, the cost of $A \bowtie B$ plus the cost of C $\bowtie D$ plus the cost of joining their results may be rnore than the cost of the cheapest left-deep plan. However, the time taken is the titne for the 1nore expensive of $A \bowtie$ B and $C \bowtie$ 1) plus the titne to join their results. Tlhis tirHe may be less than the tirne taken by the cheapest left-deep plan. This observation suggests that a parallelizing optirnizer should not restrict itself to left-deep trees and should also consider ***bushy*** trees, which significantly enlarge the space of plans to be considered.

Finally, a nurnber of pararneters, such as available buffer space and the nUl1.1-bel' of free processors, are known only at run-tirne. rrhis comnlent holds in a rnultiuser environrnent even if only sequential plans are considered; a rnultiuser environrnent is a sirnple instance of interquery parallelisrn.

22.6 INTRODUCTION TO DISTRIBUTED DATABASES

As we observed earlier, data in a distributed database systern is stored across several sites, and each site is typically rnanaged by a DBMS that can run independent of the other sites. The classical view of a distributed database systern is that the systcrn should rnake the irnpact of data distribution transparent. In particular, the following properties are considered desirable:

- Distributed Data Independence: Users should be able to ask queries without specifying where the referenced relations, or copies or fragrnents of the relations, are located. This principle is a natural extension of physical and logical data independence; we discuss it in Section 22.8. Further, queries that span rnultiple sites should be optirnized systcrnatically in a cos1,-based rnanner, taking into account COllllnunication costs and differences in local cornpntation costs. We discuss distributed query optirnization in Section 22.10.

- Distributed **Transaction** Atolnicity: Users should be able to write transactions that access and update data at several sites just as they would write transactions over purely local data. In particular, the effects of a transaction across sites should continue to be atornic; that is, all changes persist if the transaction cOllnuits and none persist if it aborts. We discuss this distributed transaction processing in Sections 22.11, 22.13, and 22.14.

AJthough rnost people would agree that these properties are in general clesirable, in certain situations, such as when sites are connected by a slow long-distance network, these properties are not efficiently achievable. Indeed, it has been argued that when sites are globally distributed, these properties are not even desirable. The argurnerlt essentially is that the adrninistrative overhead of supporting a systern with distributed data independence and transaction atomicity-:···in effect, coordinating all activities across all sites to support the view of the whole as a unified collection of data—is prohibitive, over and above DBMS perfo1'rnanc8 considerations.

}(eep these remarks about distributed databases in rnind as we cover the topic in rno1'e detail in the rest of this chapter. There is no real consensus on what the design objectives of distributed databases should be, and the field is evolving in response to users' needs.

22.6.1 Types of Distributed Databases

If data is distributed but all servers run the sarne DBMS software,. we have a **homogeneous distributed database system.** If different sites run under the control of different DBMSs, essentially autonorllously, and are connected sOlllehow to enable access to data from rnultiple sites, we have a **heterogeneous distributed database system,** also referred to as a **multidatabase system.**

The key to building heterogeneous systelTIS is to have well-accepted standards for **gateway protocols.** A gateway protocol is an API that exposes DBMS functionality to external applications. Examples include ODBC and JDBC (see Section 6.2). By accessing database servers through gateway protocols, their differences (in capability, data fonnat, etc.) are rnasked, and the differences between the different servers in a distributed system are bridged to a large degree.

Gateways are not a panacea, however. They add a layer of processing that can be expensive, and they do not cornpletely mask the differences arllong servers. For example, a server Illay not be capable of providing the services required for distributed transaction rnanagernent (see Sections 22.13 and 22.14), and even if it is capable, standardizing gateway protocols all the way down to this level of interaction poses challenges that have not yet been resolved satisfactorily.

Distributed data rnanagcrnent, in the final analysis, cornes at a significant cost in terulS of performance, software cOlllplexity, and adrninistration difficulty. rrhis observation is especially true of heterogeneous SystCIllS.

22.7 DISTRIBUTED DBMS ARCHITECTURES

Three alternative approaches are used to separat,e functionality across different DBMS-related processes; these alternative distributed])131VI8 architectures are ca,lled *Client-Server*, *Collaborating Server*, and *Middleware*.

22.7.1 Client-Server Systems

A Client-Server systelIl has one or mo1'e client processes and one or rnore server processes, and a client process can send a query to anyone server process. Clients are responsible for user-interface issues, and servers rnanage data and execute transactions. Thus, a client process could run on a personal cornputer and send queries to a server running on a 11lainframe.

This architecture has becorne very popular for several reasons. First, it is relatively sinlple to irnplernent due to its clean separation of functionality and because the server is centralized. Second, expensive server rnachines are not underutilized by dealing with lllundane user-interactions, which are now relegated to inexpensive client machines. Third, users can run a graphical user interface that they are familiar with, rather than the (possibly unfalniliar and unfriendly) user interface on the server.

While writing Client-Server applications, it is inlportant to remember the boundary between the client and the server and keep the communication between therll as set-oriented as possible. In particular, opening a cursor and fetching tuples one at a time generates many rnessages and should be avoided. (Even if we fetch several tuples and cache them at the client, rnessages 11lUSt be exchanged when the cursor is advanced to ensure that the current row is locked.) Techniques to exploit client-side caching to reduce comlnunication overhead have been studied extensively, although we do not discuss them further.

22.7.2 Collaborating Server Systems

The (;lient-Server architecture does not allow a single query to span rnultiple servers because the client process would have to be capable of breaking such a query into appropriate subqueries to be executed at different sites and then piecing together the answers to the subqueries. The client process would therefore be quite cOlnplex, and its capabilities would begin to overlap with the server; distinguishing between clients and servers becornes harder. EliIninating this distinction leads us to an alternative to the Client-Server architecture: a Collaborating Server systenl. We can have a collection of database servers, each capable of running tra,nsactions against local data, which cooperatively execute transactions spanning rnultiple servers.

When a server receives a query that requires access to data at other servers, it generates appropriate subqueries to be executed by other servers and puts the results together to COIllpute answers to the original query. Ideally, the decomposition of the query should be done using cost-based optinlization, taking into account the cost of network COlnnlunication as well as local processing costs.

22.7.3 Middleware Systems

The Middleware architecture is designed to allow a single query to span rnul-tiple servers, without requiring all database servers to be capable of rnanaging such nlulti-site execution strategies. It is especially attractive when trying to integrate several legacy systerns, whose basic capabilities cannot be extended.

The idea is that we need just one database server capable of rnanaging queries and transactions spanning nlultiple servers; the renlaining servers need to han-dle only local queries and transactions. We can think of this special server as a layer of software that coordinates the execution of queries and transactions across one or more independent database servers; such software is often called **middleware.** The middleware layer is capable of executing joins and other relational operations on data obtained froln the other servers but, typically, does not itself maintain any data.

22.8 STORING DATA IN A DISTRIBUTED DBMS

In a distributed DBMS, relations are stored across several sites. Accessing a relation stored at a renlote site incurs message-passing costs and, to reduce this overhead, a single relation lnay be *partitioned* or *fragrnented* across several sites, with fragrnents stored at the sites where they are most often accessed or *replicated* at each site where the relation is in high demand.

22.8.1 Fragmentation

Fragrnentation consists of breaking a relation into srnaller relations or frag-rnents and storing the fragrnents (instead of the relation itself), possibly at different sites. In horizontal fragmentation, each fragrnent consists of a subset of *rows* of the original relation. In vertical fragluentation, each frag-rllent consists of a subset of *columns* of the original relation. IIorizontal and verticaJ fragrnents are illustrated in Figllre 22.4.

Typically, the tuples that belong to a given horizontal fragrnent are identified by a selection query; for exarnple, crnployee tuples Blight be organized into fragments by city, with all enlployees in a given city assigned to the sanie frag-rnent. rThe horizontal fragrnent shown in Figure 22.4 corresponds to Chicago. ` ' storing fragrncnts in the database site at the corresponding city, we a,chieve :ality of reference—Chicago data is 1no8t likely to be updated and queried

TID	eid	name	city	age	sal
t1	53666	Jones	Madras	18	35
t2	53688	Snlith	Chicago	18	32
t3	53650	Sluith	Chicago	19	48
t4	53831	Madayan	Bombay	11	20
t5	53832	Guldu	BOlnbay	12	20

Vertkal Fragment Horizontal Fragment

Figure 22.4 Horizontal and Vertical Fragmentation

fronl Chicago, and storing this data in Chicago rnakes it local (and reduces cornrnunication costs) for nlost queries. Sinlilarly, the tuples in a given vertical fragrnent are identified by a projection query. The vertical fragrnent in the figure results frorn projection on the first two columns of the ernployees relation.

When a relation is fragrnented, we lllust be able to recover the original relation fronl the fragrnents:

- Horizontal Fragmentation: The union of the horizontal fragments rnust be equal to the original relation. Fragrnents are usually also required to be disjoint.
- Vertical Fragrnentation: 'The collection of vertical fragrnents should be a lossless-join deccnnposition, as per the definition in Chapter 19.

To ensure that a vertical fragrnentation is lossless-join, systeuls often assign a unique tuple ieI to each tuple in the original relation, as shown in Figure 22.4, and attach this id to the projection of the tuple in each fragrnent. If we think of the original relation as containing an addit.iC)11al tuple-id field that is a key, this field is added to each vertical fragrnent. Such a decol11position is guaranteed to be lossless-join.

In general, a relation can be (horizontally or vertically) fragrnented, a.nd each resulting fragrnent can be further fragnlented. For sirnplicity of exposition, in the rest of this chapter, we assume that fragrnents are not recursively partitioned in this rnanner.

22.8.2 **Replication**

Replication Incaus that we store several copies of a relation or relation frag-rnent. An entire relation can be replicated at one or rnore sites. Sirnila,rly, one or 1110re fragrncnts of a relation can be replicated at other sites. For example, if a relationR is fragrnented into $R1$, $R2$, and $R3$, there nlight be just one copy of $R1$, whereas $R2$ is replicated at two other sites and $R3$ is replicated at all sites.

The rnotivation for replication is twofold:

- Increased Availability of Data: If a site that contains a replica goes down, we can find the sarne data at other sites. Sirnilarly, if local copies of rerllote relations are available, we are less vulnerable to failure of COllnnu-nication links.

- Faster Query Evaluation: Queries can execute faster by using a local copy of a relation instead of going to a rernote site.

The two kinds of replication, called *synchronous* and *asynchronous* replication, differ prirnarily in how replicas are kept current when the relation is rnodified (see Section 22.11).

22.9 DISTRIBUTED CATALOG MANAGEMENT

Keeping track of data distributed across several sites can get cornplicated. We rnust keep track of how relations are fragrnented and replicated------that is, how relation fragrnents are distributed across several sites and where copies of frag-rnents are stored—in addition to the lIsuaJ seherna, authorization, and statisti-cal inforrnation.

22.9.1 Naming Objects

If a relation is fragruented and replicated, we rnust be able to uniquely identify each replica of each fragnlent. Generating such unique narnes requires sorne care. If we use a global narne-server to assign globally unique narnes, local autonomy is cornprornised; we 'want (users at) each site to be able to assign names to local objects without reference to names systernwide.

The usual solution. to the naTning problenl is to use names consisting of several fields. 1;'01' example, we could have:

- A *local name* field, 'which is the name assigned locally at the site\vhere the relation is created. T\vo objects at different sites could have the saIne local name, but two objects at a given site cannot have the saIne local narne.
- A *birth site* field, which identifies the site where the relation was created, and where il1fofrnation is ruaintained about all fragruents and replicas of the rela.tion.

These two fields identify a relation uniquely; we call the cornbination a global relation nanle. To identify a replica (of a relation or a relation fragnlent) ,we take the global relation narne and add a *replica-id* field; we call the cornbination a global replica narrle.

22.9.2 Catalog Structure

A centralized systern catalog can be used but is vulnerable to failure of the site containing the catalog. An alternative is to rnaintain a copy of a global system catalog,which describes all the data at every site. Although this approach is not vulnerable to a single-site failure, it comprornises site autonorny, just like the first solution, because every change to a local catalog rnust now be broadcast to all sites.

A better approach, which preserves local autonoruy and is not vulnerable to a single-site failure, was developed in the R* distributed database project, which was a successor to the Systerl1 R project at IBlV!. Each site ruaintains a local catalog that describes all copies of data stored at that site. In addition, the catalog at the birth site for a relation is responsible for keeping track of where replicas of the relation (in general, of fragnlents of the relation) are stored. In particular, a precise description of each replica's contents—a list of colurl1ns for a vertical fragrnent or a selection condition for a horizontal fragruentis stored in the birth site catalog. Whenever a new replica is created or a replica is rnoved across sites, the inforrnation in the birth site catalog for the relation HUlst be updated.

To locate a relation, the catalog; at its birth site lnust be looked up. This catalog inforrnation can be ca.,ched at other sites for quicker access, but the cached inforrnation Inay becolue out of date if, for cxarnple, a fragrnent is rnoved. We would discover that the locally cached inforrnation is out of date when \ve use it to access the relation, and at that point, we rll11st update the cache b.y looking up the catalog at the birth site of the relation. (The birth site of a relation is recorded in each local cache that describes the relation, and the birth site never changes, even if the relation is rnoved.)

22.9.3 Distributed Data Independence

Distributed data independence lueans that users should be able to write queries \vithout regard to ho\v a relation is fragrnented or replicated; it is the responsibility of the DBMS to cornpute the relation as needed (by locating suitable copies of fragrnents, joining the vertical fragrnents, and taking the union of horizontal fragn1cnts).

In particular, this property irnplies that users should not have to specify the full nalne for the data objects accessed while evaluating a query. Let us see how users can be enabled to access relations without considering how the relations are distributed. The *local narne* of a relation in the systeln catalog (Section 22.9.1) is really a c01l1bination of a *user narne* and a user-defined *relation narne.* 'Users can give whatever names they wish to their relations, without regard to the relations created by other users. When a user writes a prograrn or SQL statelnent that refers to a relation, he or she sirnply uses the relation narne. The DBMS adds the user narne to the relation narne to get a local narne, then adds the user's site-id as the (default) birth site to obtain a global relation narne. By looking up the global relation narne-- –in the local catalog if it is cached there or in the catalog at the birth site-·the DBMS can locate replicas of the relation.

A user Il1ay want to create objects at several sites or refer to relations created by other users. To do this, a user can create a synonym for a global relation narne' llsing an SQL-style cOl1unand (although such a corl1rnand is not currently part of the SQL:1999 standard) and subsequently refer to the relation using the synonyrn. For each user known at a site, the DBMS maintains a table of synonynls as part of the systern catalog at that site and uses this table to find the global relation narne. Note that a user's prograrl1 runs unchanged even if replicas of the relation are rlloved, because the global relation narne IS never changed until the relation itself is destroyed.

lJsers rnay want to run queries against specific replicas, especially if asynchronous replication is used. To support this, the synonyrn Inechanisrn can be adapted to also allo\v users to create synon.yrl1S for global replica, names.

22.10 DISTRIBUTED QUERY PROCESSING

We first discuss the issues involved in evaluating relational algebra operations in a distrilnlted database through exalnples and then outline distributed query optiInization. Consider the following t\vo relations:

Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Reserves(*sid:* integer, *bid:* integer, *day:* date, *rname:* string)

As in Chapter 14, assurne that each tuple of R.eserves is 40 bytes long, tha.t a page can hold 100 Reserves tuples, and that we have 1000 pages of such tuples. Sirnilarly, assurne that each tuple of Sailors is 50 bytes long, that a page can hold 80 Sailors tuples, and that we have 500 pages of such tuples.

To estimate the cost of an evaluation strategy, in addition to counting the nurnber of page IjC)s, \ve Blust count the nurnher of pages sent frorH one site to another because corllrIlunication costs are a significant cornponent of overall cost in a distributed database. We rnust also change our cost rnodel to count the cost of shipping the result tuples to the site where the query is posed frOIn the site where the result is assernbled! In this chapter, we denote the time taken to read one page from disk (or to write one page to disk) as t_d and the tiIne taken to ship one page (from any site to another site) as t_s.

22.10.1 Nonjoin Queries in a Distributed DBMS

Even sirnple operations such as scanning a relation, selection, and projection are affected by fragmentation and replication. Consider the following query:

```
SELECT  S.age
FROM    Sailors S
WHERE   S.rating > 3 AND S.rating < 7
```

Suppose that the Sailors relation is horizontally fragruented, with all tuples having a rating less tha,n 5 at Shanghai and all tuples having a rating greater than 5 at rrokyo.

TheDBl\/IS nn.lst answer this query by evaluating it at both sites and taking the union of the ans\vers. If the SELECT clause contained AVG *(S. age),* cOIn-bining the answers could not be done by sirnply taking the union···-····theDBMS rnust cornpute the sUIn and count of *age* values at the two sites and use this infonna,tion to cornpute the average age of all sailors.

If the WHERE clause contained just the condition 5'.*rating* > *6*, on the other ha,ud, the I)BI\/IS should recognize that this query could be answered by just executing It at Tnkyo.

As another example, suppose that the Sailors relation, were vertically frag-mented, with the *sid* and *rating* fields at Sha.nghai a,nd the *sname* and *age* fields at rrokyo. No field is stored at both sites. This vertical fragmentation would therefore be a lossy decornposition, except that a field containing the id of the corresponding Sailors tuple is included by the DBMS in both fraglnents! Now, the DBMS has to reconstruct the Sailors relation by joining the tVO fraglnents on the COHlllon tuple-id field and execute the query over this reconstructed relation.

Finally, suppose that the entire Sailors relation were stored at both Shanghai and Tokyo. We could answer any of the previous queries by executing it at either Sha,nghai or Tokyo. Where should the query be executed? This depends on the cost of shipping the answer to the query site (which rnay be Shanghai, Tokyo, or SOllIe other site) as well as the cost of executing the query at Shanghai and at Tokyo—the local processing costs lnay differ depending on what indexes are available on Sailors at the two sites, for exaluple.

22.10.2 Joins in a Distributed DBMS

Joins of relations at different sites can be very expensive, and we now consider the evaluation options that IllUst be considered in a distributed environrnent. Suppose that the Sailors relation were stored at London, and the Reserves relation were stored at Paris. We consider the cost of various strategies for cOlnputing *Sailor'S* $\bowtie$ *Reserves*.

Fetch As Needed

We could do a page-oriented nested loops join in Loudon with Sailors as the outer, and for each Sailors page, fetch all Reserves pages frorn Paris. If we cache the fetched Reserves pages in London until the join is complete, pages are fetched only once, but aSSU111e that H,eservcs pages are not cached, just to see how bad things can get. (The situation can get rnuch worse if we use a tuple-oriented nested loops join!)

rrhe cost is $500t_d$ to scan Sailors plus, for each Sailors page, the cost of seallning and shipping all of Reserves, which is $1000(t_d + t_s)$. The total cost is therefore $500t_d + 500{,}000(t_d + t_s)$.

In addition, if the query was not sllbrnittccl at the London site, we rnust add the cost of shipping the result to the query site; this cost depends on the size of the result. Because *sid* is a key for Sailors, the nurnber of tuples in the result is 100,000 (the rnunber 0f tuples in Reserves) and each tuple is $40 + 50 = 90$ bytes long; thus $4000/90 = 44$ result tuples fit on a page, and the result size is $100{,}000/44{=}2273$ pages. The cost of shipping the answer to another site, if necessary, is $2273\ t_s$. In tIle rest of this section, we assume that the query is posed at the site where the result is computed; if not, the cost of shipping the result to the query site Blust be added to the cost.

In this exarnple, observe that, if the query site is not London or Paris, the cost of shipping the result is greater than the cost of shipping both Sailors and Reserves to the query site! Therefore, it would be cheaper to ship both relations to the query site and COlllpute the join there.

Alternatively, we could do an index nested loops join in London, fetching all Inatching Reserves tuples for each Sailors tuple. Suppose we have an uncIustered hash index on the *sid* columnn of Ileserves. Because there are 100,000 Ileserves tuples and 40,000 Sailors tuples, each sailor has on average 2.5 reservations. The cost of finding the 2.5 Ileservations tuples that lllatch a given Sailors tuple is $(1.2 + 2.5)t_d$, assluning 1.2 l/Os to locate the appropriate bucket in the index. The total cost is the cost of scanning Sailors plus the cost of finding and fetching nlatching Reserves tuples for each Sailors tuple, $500t_d + 40,000(3.7t_d + 2.5t_s)$.

Both algorithIns fetch required Reserves tuples from a remote site as needed. Clearly, this is not a good idea; the cost of shipping tuples dominates the total cost even for a fast network.

Ship to One Site

We can ship Sailors from London to Paris and carry out the join there, ship Reserves to London and carry out the join there, or ship both to the site where the query was posed and cornpute the join there. Note again that the query could have been posed in London, Paris, or perhaps a third site, say, Tirnbuktu!

1'he cost of scanning and shipping Sailors, saving it at Paris, then doing the join at Paris is $500(2t_d + t_s) + 4500t_d$, assurning that the version of the sort-rnerge join described in Section 14.10 is used and we have an adequate nurnber of buffer pages. In the rest of this section we aSSUInc that sort-Inerge join is the join rnethod used when both relations are at the saIne site.

The cost of shipping Reserves and doing the join at London is $1000(2t_d + t_s) + 4500t_d$.

Senlijoins andBloomjoins

Consider the strategy of shipping Reserves to Londo.l1 and cornputing the join at London. Some tuples in (the current instance of)H,cserves do not join with an.y tuple in (the current instance of) Sailors. If we could somehow identify

Reserves tuples that are guaranteed not to join with any Sailors tuples, we could avoid shipping thern.

Two techniques, *Semijoin* and *Bloomjoin*, have been proposed for reducing the number of lleserves tuples to be shipped. The first technique is called Semijoin. The idea is to proceed in three steps:

1. At London, cornpute the projection of Sailors onto the join colurnns (in this case just the *sid* field) and ship this projection to Paris.
2. At Paris, cornpute the natural join of the projection received frorn the first site with the R,eserves relation. The result of this join is called the reduction of R,eserves with respect to Sailors. Clearly, only those Reserves tuples in the reduction will join with tuples in the Sailors relation. Therefore, ship the reduction of Reserves to London, rather than the entire Reserves relation.
3. At London, cornpute the join of the reduction of R,eserves with Sailors.

Let us compute the cost of using this technique for our example join query. Suppose we have a straightforward irnplernentation of projection based on first scanning Sailors and creating a telnporary relation with tuples that have only an *sid* field, then sorting the temporary and scanning the sorted ternporary to eliminate duplicates. If we assurne that the size of the *sid* field is 10 bytes, the cost of projection is $500t_d$ for scanning Sailors, plus 100td for creating the ternporary, plus $400t_d$ for sorting it (in two passes), plus IOOtc! for the final scan, plus $100t_d$ for writing the result into another tcrnporary relation; a total of $1200t_d$. (Because *sid* is a key, no duplicates need be elirninated; if the optiInizer is good enough to recognize this, the cost of projection is just (500 + 100)td.)

The cost of cornputing the projection and shipping it to Paris is therefore *1200ld* + $100t_s$. The cost of c(nnputing the reduction of R.eserv8s is 3 . (100 + 10(0) = $3300t_d$, assurning that sort-rnerge join is used. (The cost does not reflect that the projection of Sailors is already sorted; the cost would decrease slightly if the refined sort-Inerge join exploited this.)

What is the size of the reduction? If every sailor holds at least one reservation, the reduction includes every tuple of R,eserves! The effort invested in shipping the projection and reducing Reserves is a total waste. Indeed, because of this observation, we note that Sernijoin is especially useful in conjunction with a selectiol1 on one of the relations. For example, if we want to cornpute the join of Sailors tuples with a *rating* greater than 8 with the Reserves rela.tion, the size of the projection on *sid* for tuples that satisfy the selection would be just 20 percent of tIle original projection, that is, 20 pages.

Let us now continue the example join, 'with the assurnption that we have the additional selection on *rating.* (The cost of cornputing the projection of Sailors goes down a bit, the cost of shipping it goes down to $20t_s$, and the cost of the reduction of Reserves also goes down a little, but we ignore these reductions for sirnplicity.) \Ve assurne that; only 20 percent of the Reserves tuples are included in the reduction, thanks to the selection. Hence, the reduction contains 200 pages, and the cost of shipping it is $200t_s$.

Finally, at London, the reduction of I{eserves is joined with Sailors, at a cost of $3 \cdot (200 + 500) =$ *21100td.* Observe that there are over 6500 page I/Os versus about 200 pages shipped, using this join technique. In contrast, to ship R,eserves to London and do the join there costs $1000t_s$ plus 4500td. With a high-speed. network, the cost of Sernijoin Illay be n10re than the cost of shipping Reserves in its entirety, even though the shipping cost, itself is rnueh less ($200t_s$ versus I0O0t_s)·

The second technique, called Bloomjoin, is quite sirnilar. The luain difference is that a bit-vector is shipped in the first step, instead of the projection of Sailors. A bit-vector of (soinc chosen) size k is cOlnputed by hashing each tuple of Sailors into the range 0 to k - I and setting bit i to I if seHne tuple hashes to i, and 0 otherwise. In the second step, the reduction of Reserves is cOlnputed by hashing each tuple of Reserves (using the *sid* field) into the range 0 to k --*1,* using the sanle hash function used to construct the bit-vector and discarding tuples whose hash value i corresponds to a 0 bit. Because no Sailors tuples hash to such an i, no Sailors tuple can join with any R,eserves tuple that is not in the reduction.

The costs of shipping a bit-vector and reducing R,eserves using the vector are less than the corresponding costs in Sernijoin. ()n the other hand, the size of the reduction of Reserves is likely to be larger than in Sernijoin; so, the costs of shipping the reduction and joining it 'with Sailors are likely to be higher.

Let us estirnate the cost of this approach. rrhe cost of cornputing the bit-vector is essentially the cost of scanning Sailors, \vhich is *500td.* The cost of sending the bit-vector depends on the size we choose for the bit-vector, 'which is certainly sInaJler than the size of the projection; we take this cost to be 201:$_8$, for concreteness. The cost of reducing Reserves is just the cost of scanning Reserves, $1000t_d$. T'he size of the reduction of Reserves is likely to be about the saIne as or a little larger than the size of the reduction in tIle Scrnijoin approach; instea,d of 200, we will take tllis size to be 220 pages. (We assume that the selection on Sailors is included, to pennit a direct cOInparison \vith the cost of Scrnijoin.) 'I'he cost of shipping the reduction is therefore $220t_s$' The cost of the final join at London is $3 \cdot (500 + 220) =$ *2160td.*

rrhus, in cornparison to Semijoin, the shipping cost of this approach is about the same, although it could be higher if the bit-vector were not as selective as the projection of Sailors in terms of reducing Reserves. Typically, though, the reduction of Reserves is no l110re than 10 to 20 percent larger than the size of the reduction in SClnijoin. In exchange for this slightly higher shipping cost, Bloornjoin achieves a significantly lower processing cost: less than $3700t_d$ versus rnore than 6500td for SClnijoin. Indeed, Bloornjoin has a lower I/C) cost and a lower shipping cost than the strategy of shipping all of R,eserves to London! These nurnbers inclicatewhy Bloollljoin is an attractive distributed join rnethod; but the sensitivity of the rnethod to the effectiveness of bit-vector hashing (in reducing Reserves) should be kept in rnind.

22.10.3 Cost-Based Query Optimization

We have seen how data distribution can affect the inlplernentation of individual operations, such as selection, projection, aggregation, and join. In general, of course, a query involves several operations, and optirnizing queries in a distributed database poses the following additional challenges:

- CornrIlunication costs lllUSt be considered. If we have several copies of a relation, we HlllSt also decide which copy to use.
- If individual sites are run under the control of different DBl\JlSs, the autonolny of each site HUlst be respected while doing global query planning.

Query optiInization proceeds essentially as in a centralized DBMS, as described in Chapter 12, with infornation about relations at rernote sites obtained fron1 the systeln catalogs. ()f course, there are nlore alternative lllethods to consider for each operation (e.g., consider the new options for distributed joins), and the cost rnetric ruust account for cornrnunication costs as \vell, but the overall planning process is essentially unchanged if we take the cost rnetric to be the total cost of all operations. (If we consider response tilne, the fact that certain subqueries can be carried out in parallel at different sites \vould require us to change the optirnizer as per the discussion in Section 22.5.)

In the overall plan, local rnanipulatiol1 of relations at the site where they are stored (to corllpute an interrnediate relation to be shipped elsewhere) is encapsulated into a *suggested* local plarl. The overall plan includes several such local plans, \vhichwe can think of as subqueries executing at different sites. \:Vhile generating the global plan, the suggested local plans provide realistic cost estimates for the cornputatioll of the interrnediate relations; the suggested local plans are constructed by the optirnizer rnainly to provide these local cost estiIuates. A site is free to ignore the local plan suggested to it if it is able to find a cheaper plan by llsing more current infonnation in the local catalogs. Thus, site autonomy is respected in the optimization and evaluation of distributed quenes.

22.11 UPDATING DISTRIBUTED DATA

The classical view of a distributed DBMS is that it should behave just like a centralized DBMS froul the point of view of a user; issues arising froln distribution of data should be transparent to the user, although, of course, they rnu8t be addressed at the irnplernentation level.

With respect to queries, this view of a distributed DBIVIS Ineans that users should be able to ask queries \vithout worrying about how and where relations are stored; we have already seen the irnplications of this requirernent on query evaluation.

With respect to updates, this view rneans that transactions should continue to be atornic actions, regardless of data fragrnentation and replication. In particular, all copies of a rnodified relation must be updated before the rnodifying transaction cornn1its. We refer to replication with this sernantics as synchronous replication; before an update transaction cOHllnits, it synchronizes all copies of rnodified data.

An alternative approach to replication, called asynchronous replication, has corne to be widely usee1 in eornrnercial distributed DBlVISs. Copies of a rnodified relation are updated only periodically in this approach, and a transaction that reads different copies of the sarne relation rnay see different values. T'hus, asynchronous replication cornprolnises distributed data independence, but it can be iInplernented 1110re efficiently than synchronous replication.

22.11.1 Synchronous Replication

There are two basic techniques for ensuring that transactions see the same value regardless of\vhich copy of an object they access. In the first technique, called voting, a transaction Inust write H" lnajority of copies to rnodify an ol)ject and read at least enough copies to rnake sure that one of the copies is current. For exanlple, if there are 10 copies and 7 copies are written by update transactions, then at least 4 copies rnust be read. Eac:h copy has a version nurnber, and the copy with the highest version rllunber is current. This technique is not attra,ctive in rnost situations because reading an ol)ject reqllires reading rnultiple copies; in rnost applications, objects are read rnuch 1n01'e frequently than they are updated, and efficient performance on reads is very irnportant.

In the second technique, called **read-any write-all**, to read an object, a traJ1Saction can read anyone copy, but to write an object, it Inust write all copies. Reads are fast, especially if we have a local copy, but 'writes are slower, relative to the first technique. This technique is attractive when reads are rnuch rnore frequent than writes, and it is usually adopted for irnplernenting synchronous replication.

22.11.2 Asynchronous Replication

Synchronous replication COines at a significant cost. Before an update transaction can corn1'nit, it rnust obtain exclusive locks on all copies—assuming that the read-any write-all technique is used· ···ofrnodified data. The transaction Inay have to send lock requests to rernote sites and wait for the locks to be granted, and during this potentially long period, it continues to hold all its other locks. If sites or connnunication links fail, the transaction cannot cOInrnit until all the sites at which it has rnodified data recover and are reachable. Finally, even if locks are obtained readily and there are no failures, connnitting a transaction requires several additional rnessages to be sent as part of a *commit protocol* (Section 22.14.1).

For these reasons, synchronous replication is undesirable or even unachievable in 111any situations. Asynchronous replication is gaining in popularity, even though it allows different copies of the saIne object to have different values for short periods of tinlC. This situation violates the principle of distributed data independence; users 11Ulst be aware of which copy they are accessing, recognize that copies are brought up-to-date only periodically, and live with this reduced level of data consistency. Nonetheless, this seeIns to be a practical C0l11pr0l11ise that is acceptable in rnany situations.

Primary Site versus **Peer-to-Peer** Replication

A.synchronous replication C01nes in two flavors. In **primary site** asynchronous replication, one copy of a relation is designated the **primary** or **luaster** copy. H.eplicas of the entire relation or fragrnents of the relation can be created at other sites; these are **secondary** copies, and unlike tIle primary copy, they cannot be updated. A conUIlon IneclIallislIl for setting up primary and secondary copies is that users first **register** or **publish** the relation at the primary site and subsequently **subscribe** to a fragment of a registered relation fron1 another (secondary) site.

In **peer-to-peer** asynchronous replication, 111()re than one copy (although perhaps rIot all) can be designated as updatable, that is, a 1'naste1' copy. In addition to propagating changes, a **conflict resolution** strategy must be used to deal with conflicting changes Inade at different sites. For example, .Joe's age may be changed to 35 at one site and to 38 at another. Which value is 'correct'? Many luore subtle kinds of conflicts can arise in peer-to-peer replication, and in general peer-to-peer replication leads to ad hoc conflict resolution. Some special situations in which peer-to-peer replication does not lead to conflicts arise quite often, and in such situations peer-to-peer replication is best utilized. For example:

- Each 1naster is allo\ved to update only a fragrnent (typically a horizontal frag1nent) of the relation, and any two fragrnents updatable by different !llasters are disjoint. For example, it rIlay be that salaries of Gerrnan erll-ployees are updated only in Frankfurt, and salaries of Indian ernployees are updated only in 1\1adras, even though the entire relation is stored at both Frankfurt and Madras.
- Updating rights are held by only one rnaster at a tillIe. For example, one site is designated a *backup* to another site. Changes at the Iuaster site are propagated to other sites and updates are not allowed at other sites (including the backup). But, if the Iuaster site fails, the backup site takes over and updates are now perrnitted at (only) the backup site.

We will not discuss peer-to-peer replication further.

Implementing Primary Site Asynchronous Replication

The Inain issue in irnpler11enting prilnary site replication is deterrnining how changes to the prirnary copy are propagated to the secondary copies. Changes are usually propagated in two steps, called *Capture* and *Apply.* Changes rnade by cOHnnitted transactions to the prirnary copy are s()Jnehow identified during the Capture step and subsequently propagated to secondary copies during the Apply step.

In contrast to synchronous replication, a transacti.on that rnodifies a replicated relation directly locks and changes only the prirnary copy. It is typically C0111-rnitted long before the Apply step is carried out. Systcrnsvary considerably in their ilnplernentation of these steps. We present an overview of some of the alternatives.

Capture

rrlle Capture step is implemented using one of two approaches. In log-based Capture, the log luainta,inecl for recovery purposes is used to generate a record of updates. Basically, when the log tail is written to stable storage, all log records that affect replicated relations are also written to a separate change data table (eDT). Since the transaction that generated the update log record may still he active when the record is\vritten to the CDT, it may subsequently abort. lJpdate log records written by transactions that subsequently abort 1l1USt l)e rcrnoved frorl1 the eDT to obtain a strearll of updates due (only) to conlln,itted transactions. This streanl can be obtained as part of the Capture step or subsequently in the Apply step if conunit log records are added to the eDT; for concreteness, we aSSUlne that the cornruitted update streall1 is obtained as part of the Capture step and that the CDT sent to the Apply step contains only update log records of corl1ruitted transactions.

In procedural Capture, a procedure autornatically invoked by the DBMS or an application progra,lIl initiates the Capture process, which consists typically of taking a snapshot of the prirnary copy. A snapshot is just a copy of the relation as it existed at sorne instant in tirne. (A procedure that is autoluatically invoked by the DBMS, such as the one that initiates Capture, is called a *trigger*. We covered triggers in Chapter 5.)

Log-based Capture has a s111aller overhead than procedural Capture and, because it is driven by changes to the data, results in a slua11er delay between the tirne the prirnary copy is changed and the ti111e that the change is propagated to the secondary copies. (Of course, this delay also depends on ho\v the Apply step is implelnented.) In particular, only changes are propagated, and related changes (e.g., updates to two tables with a referential integrity constraint between thern) are propagated together. The disadvantage is that ilnpleluenting log-based Capture requires a detailed understanding of the structure of the log, which is quite systern specific. Therefore, a· vendor cannot easily implement a log-based Capture rnechanisrn that will capture changes rnade to data in another vendor's DBMS.

Apply

fI'he Apply step takes the changes collected by the Capture step, which are in the CDT table or a snapshot, and propagates 1,h81n to the secondary copies. This can be done by having the prirnary site continuously send the CDT or periodically requesting (the latest portion of) the CDT or a snapshot frorH the prirnary site. Typically, each secondary site runs a copy of the J\pply process and 'pulls' the changes in the eDT fronl the prirnary site using periodic requests. The interval between such requests can be controlled by a timer or a user's application prograrl1. ()nce the changes are avail(1)le at the secondary site, they can be applied directly to the replica.

In sorne systems, the replica Heed not be just a frag1Ilent of the original relation· it can be a view defined using SQL, and the replication rnechanisrn is sufficiently sophisticated to 11laintain such a view at a reillote site incrementally (by reevaluating only the part of the vie\v affected by changes recorded in the CI)T).

Log-based Capture in conjunction with continuous Apply rninirnizes the delay in propagating changes. It is the best cor11bination in situations where the primary and secondary copies are both used as part of an operational DBMS and replicas must be as closely synchronized with the prinlary copy as possible. Log-based Capture with continuous Apply is essentially a less expensive substitute for synchronous replication. Procedural Capture and application-driven Apply offer the 11l0St flexibility in processing source data and changes before altering the replica; this flexibility is often useful in data warehousing applications where the ability to 'clean' and filter the retrieved data is 1110re important than the currency of the replica.

Data Warehousing: An Example of Replication

Cornplex decision support queries that look at data from Illultiple sites are becoming very inlportant. The paradigrn of executing queries that span r11ultiple sites is sirnply inadequate for perfornlance reasons. One way to provide such complex query support over data froln rllultiple sources is to create a copy of all the data at SaIne one location and use the copy rather than going to the individual sources. Such a copied collection of data is called a data warehouse. Specialized systelIls for building, rnaintaining, and querying data warehouses have becolne irnportant tools in the rnarketplace.

Data warehouses can be seen as one instance of asynchronous replication, in 'which copies are updated relatively infrequently. When we talk of replication, we typically rIlCal1 copies Inaintained under the control of a single DBMS, \vhereaswith data \varehousing, the original data rnay be on different software platforrns (including database systems and as file systerIls) and even l)clong to different organizations. This distinction, 110\VeVer, is likely to becoine blurred as vendors adopt luore 'open' strategies to replication. For exarnple, sorne products already support the maintenance of replicas of relations stored in one vendor's DBMS in al10ther vendor's DBMS.

We 110te that data warehousing involves rnore than just replication. We discuss other aspects of data warehousing in Chapter 2.5.

22.12 DISTRIBUTED TRANSACTIONS

In a distributed DBMS, a given transaction is subrnitted at SOIne one site, but it can access data at other sites as well. In this chapter we refer to the activity of a transaction at a given site as a subtransaction. When a transaction is subrnitted at S0111e site, the transaction rnanager at that site breaks it up into a collection of one or rnoro subtransactions that execute at different sites, subrnits theln to transaction rnanagers at the other sites, and coordinates their activity.

We now consider aspects of concurrency control and recovery that require additional attention because of data distribution. As we saw in Chapter 16, there are many concurrency control protocols; in this chapter, for concreteness, we assurne that Strict 2PL with deadlock detection is used. We discuss the following issues in subsequent sections:

- Distributed Concurrency Control: How can locks for objects stored across several sites be managed? How can deadlocks be detected in a distributed database?
- Distributed Recovery: Transaction atomicity lllUSt be ensured-·---when a transaction commits, all its actions, across all the sites at which it executes, rnust persist. Si111ilarly, when a transaction aborts, none of its actions must be allowed to persist.

22.13 DIS"fRIBUTED CONCURRENCY CONTROL

In Section 22.11.1, we described t\vo techniques for irnplernenting synchronous replication, and in Section 22.11.2, we discussed various techniques for irllplernenting asynchronous replication. rrhe choice of technique deterrnines *which* objects are to be locked. *When* locks are obtained and released is deterrnined by the concurrency control protocol.vVe now consider how lock and unlock requests are implemented in a distributed envirorllnent.

Lock rnanagernent can be distributed across sites in rnanyways:

- Centraliz,ed: A single site is in charge of handling lock and unlock requests for all objects.
- Priulary Copy: ()ne copy of each object is designated the primary copy. All requests to lock or unlock a copy of this object are handled by the lock rnanager at the site where the prirnary copy is stored, regardless of where the copy itself is stored.

- **Fully Distributed:** R,equests to lock or unlock a copy of an object stored at a site are handled by the lock lnanager at the site where the copy is stored.

The centralized schelne is vulnerable to failure of the single site that controls locking. The prirnary copy scherne avoids this problern, but in general, reading a,n object requires cornrnunicatiollwith t\VO sites: the site where the prirnary copy resides and the site where the copy to be read resides. This problern is avoided in the fully distributed 8che1nc, because locking is done at the site where the copy to be read resides. However, while writing, locks rnust be set at all sites where copies are rnoclified in the fully distributed schclne, whereas locks need be set only at one site in the other two schernes.

Clearly, the fully distributed locking scherne is the 1110st attractive schelne if reads are much more frequent than writes, as is usually the case.

22.13.1 Distributed Deadlock

One issue that requires special attention when using either priluary copy or fully distributed locking is deadlock detection. (Of course, a deadlock prevention scherne can be used instead, but we focus on deadlock detection, which is widely used.) As in a centralized DBMS, deadlocks rnust be detected and resolved (by aborting sorne deadlocked transaction).

Each site rnaintains a local waits-for graph, and a cycle in a local graph indicates a, deadlock. lIowever, there can be a deadlock even if no local graph contains a cycle. For exarnple, suppose that two sites, A and B, both contain copies of objects 01 and 02, and that the read-any write-all technique is used. $T1$, which wants to read ()1 and write 02, obtains an S lock on 01 and an X lock on 02 at Site A, then requests an X lock on 02 at Site B. $T2$, which \vants to read 02 and write 01, rneanwhilc, obtains an S lock on 02 and an X lock on 01 at Site B, then requests an X lock on ()1 at Site A. A.s Figure 22.5 illustrates, $T2$ is waiting for Tl at Site A. and Tl is waiting for $T2$ at Site 13; thus, we have a deadlock, \vhich neither site can detect based solely on its local waits-for graph.

To detect such deadlocks, a **distributed deadlock detection** algoritlun rnust be used. We describe three such algoritluns.

The first algorithrn,\vhich is centralized, consists of periodically sending all 10-cal waits-for graphs to one site that is responsible for global deadlock detection. At this site, the global waits-for graph is generated by cOlubinin.g all the local graphs; the set of nodes is the union of nodes in the local graphs, and there is

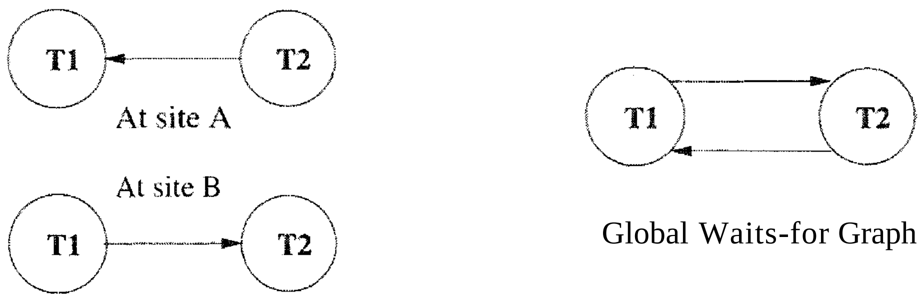


Figure 22.5 Distributed Deadlock

an edge frorn one node to another if there is such an edge in any of the local graphs.

The second algorithrn, which is hierarchical, groups sites into a hierarchy. For instance, sites r11ight be grouped by state, then by country, and finally into a single group that contains all sites. Every node in this hierarchy constructs a waits-for graph that reveals deadlocks involving only sites contained in (the subtree rooted at) this node. All sites periodically (e.g., every 10 seconds) send their local waits-for graph to the site responsible for constructing the waits-for graph for their state. The sites constructing waits-for graphs at the state level periodically (e.g., every minute) send the state waits-for graph to the site constructing the waits-for graph for their country. The sites constructing waits-for graphs at the country level periodically (e.g., every 10 rninutes) send the country waits-for graph to the site constructing the global waits-for graph. This scheme is based on the observation that l110re deadlocks are likely across closely related sites than across unrelated sites, and it puts 1110re effort into detecting deadlocks across related sites. All deadlocks are eventually detected, but a deadlock involving two different countries J.nay take a while to detect.

The third algorithrn is sirllple: If a transaction waits longer than SOIne chosen tinle-out interval, it is aborted. Although this algorithrll rnay cause rnany unnecessary restarts, the overhead of deadlock detection is (obviously!) low, and in a heterogeneous distributed database, if the participating sites cannot cooperate to the extent of sha,ring their \va,its-for graphs, it rnay be the only option.

A subtle point to note with respect to distributed deadlock detection is that delays in proI5agating local infornation rnight cause the deadlock detection algorithr11 to identify 'deadlocks' that do not really exist. Such situations, called phantoln **deadlocks**, lead to unnecessary aborts. For concreteness, we cliscuss the centralized algorithrn, although the hierarchical algorithrn suffers fr0111 the same problern.

Consider a rnodificatioll of the previous exarnple. As before, the two transactions wait for each other, generating the local \vaits-for graphs shown in Figure 22.5, and the local waits-for graphs are sent to the global deadlock-detection site. IIo\vever, $T2$ is now aborted for 1'easons other than deadlock. (For example, *T2* rnay also be executing at a third site, 'where it reads an unexpected data value and decides to abort.) At this point, the local waits-for graphs have changed so that there is no cycle in the 'true' global \vaits-for graph. However, the constructed globaJ waits-for graph \vill contain a cycle, and $T1$ Inay well be picked as the victirn!

22.14 DISTRIBUTED RECOVERY

Recovery in a distributed DBMS is rnore cornplicated than In a centralized DBMS for the following reasons:

- New kinds of failure can arise: failure of COlnnlunication links and failure of a remote site at which a subtransaction is executing.
- Either all subtransactions of a given transaction Iuust ccnnlnit or none HUlst conlnlit, and this property IIlust be guaranteed despite any cOIllbination of site and link failures. T'his guarantee is achieved using a **commit protocol**.

As in a centralized DBMS, certain actions are carried out as part of norrnal execution to provide the necessary infonnation to recover fro111 failures. A log is rnaintained at each site, and in addition to the kinds of inforrnation rnaintained in a centralized DBMS, actions taken as part of the cOlInnit protocol are also logged. The Inost widely used conunit protocol is called *Two-Phase Cornmit (2PC).* A variant called 21'C *with Presumed Abort*, which we discuss next, has been adopted as an industry standard.

In this section, we first describe the steps taken during nonnal execution, concentrating on the cOHnnit protocol, and tJlen discuss recovery fron1 failures.

22.14.1 Normal Execution and Commit I>rotocols

I)uring Donnal execution, each site rnaintains a log, and the actions of a subtransaction are logged at the site where it executes. The regular logging activity described in Chapter 18 is carried out and, in addition, a eornnlit protocol is followed to ensure that all subtra,nsa.ctions of a given transaction either cOIrnnit or H,bort uniforrnly. The transaction rnanager at the site where the transaction originated is called the **coordinator** for the transaction; transaction Inanagers at sites where its subtraJ1Sactiol1S execute are called **subordinates** (with respect to the coordinatioll of this transaction).

We no\v describe the **Two-Phase** Cornmit (2PC) protocol, in terms of the messages exchanged and the lc)g records written. When the user decides to cornrnit a transaction, the conunit cOIrunand is sent to the coordinator for the transaction. This initiates the 2PC protocol:

1. The coordinator sends a *prepare* rnessage to each subordinate.

2. When a subordinate receives a *prepare* message, it decides whether to abort or cornrnit its subtransaction. It force-writes an abort or prepare log record, and *then* sends a *no* or *yes* rnessage to the coordinator. Note that a prepare log record is not used in a centralized DBMS; it is unique to the distributed cornrnit protocol.

3. If the coordinator receives ***yes*** lnessages fro III all subordinates, it force-writes a cornmit log record and then sends a *cornrnit* rnessage to all subordinates. If it receives even one *no* rnessage or receives no response fronl SOHle subordinate for a specified titne-out interval, it force-writes an abort log record, and then sends an *abort* Inessage to all subordinates.[1]

4. When a subordinate receives an *abort* Inessage, it force-writes an abort log record, sends an *ack* Inessage to the coordinator, and aborts the subtransaction. When a subordinate receives a *cornrnit* rnessage, it force-writes a cOl1nnit log record, sends an *ack* rnessage to the coordinator, and corrunits the subtransaction.

5. After the coordinator has received *ack* rnessages frorn all subordinates, it writes an end log record for the transaction.

1:'he narne *Two-Phase Commit* reflects the fact that two rounds of rnessages are exchanged: first a voting phase, then a tennination phase, both initiated by the coordinator. ffhe basic principle is that any of the transaction tnanagel'S involved (including the coordinator) can unilaterally a,bort a transaction, \vhereas therernust be unanirnity to conuuit a transaction, When a rnessage is serlt in 2PC, it signals a decision by the sender. To ensure that this decision survives a crash at the sender's site, the log record describing the decision is always forced to stable storage *before* the rnessage is sent.

;\ transaction is ofIicially cornrnitted at the tirne the coordiIlator's cOllnnit log record reaches stable storage. Subsequent failures cannot affect the outcorne of the transaction; it is irrevocaJ)ly corrunitted. Log records\vritten to record the connnit protocol actions contain the type of the record, the transaction id, and the identity of the coordinator. *i* coordinator's conunit or abort log record also contains the identities of the subordinates.

[1] As a,n optilnization, the coordinator need not send *abort* messages to subordinates who voted *no,*

22.14.2 Restart after a Failure

When a site COUles back up after a crash, we invoke a recovery process that reads the log and processes all transactions executing the conunit protocol at the tirne of the crash. The transaction rnanager at this site could have been the coordinator for SOUle of these transactions and a subordinate for others. We do the following in the recovery process:

- If we have a cornInit or abort log record for transaction T, its status is clear; we redo or undo T, respectively. If this site is the coordinator, which can be deterrnined froru the cOlTllnit or abort log record, we rnust periodically resend—because there rnay be other link or site failures in the system—a *commit* or *abort* rnessage to each subordinate until we receive an *ack*. After we have received *acks* frorn all subordinates, we write an end log record for T.
- If we have a prepare log record for T but no conunit or abort log record, this site is a subordinate, and the coordinator can be detennined froIn the prepare record. We rllust repeatedly contact the coordinator site to determine the status of T. Once the coordinator responds with either cOlurnit or abort, we write a corresponding log record, redo or undo the transaction, and then write an end log record for T.
- If we have no prepare, cOllunit, or abort log record for transaction T, T certainly could not have voted to connuit before the crash; so we can unilaterally abort and undo T and write an end log record. In this case, we have no way to detennine whether the current site is the coordinator or a subordinate for T. flowever, if this site is the coordinator, it rnight have sent a *prepare* rnessage prior to the crash, and if so, other sites rnay have voted *yes*. If such a subordinate site contacts the recovery process at the current site, we now know that the current site is the coordinator for T, and given that there is no cOllnnit or abort log record, the response to the subordinate should be to abort T.

()bserve that, if the coordinator site for a transaction T fails, subordinates who voted *yes* cannot decide whether to conunit or abort T until the coordinator site recovers; we say that T is blocked. In principle, the active subordinate sites could cOl1nnunicate arnong thelIlselves, and if at least one of thelIl contains an abort or coinrnit log record for T, its status becornes globally known. 1"0 conununicate arnong thernselves, all subordinates nlust be told the identity of the other subordinates at the time they are sent the *prepare* message. llowever, 2PC is still vulnerable to coordinator failure durirlg recovery because even if all subordinates voted *yes*, the coordinator (who also has a vote!) may have decided to aJ)ort T, and this decision cannot be determined until the coordinator site recovers.

We covered how a site recovers fro111 a crash, but what should a site that is involved in the cOIIunit protocol do if a site that it is cornrl1unicating with fails? If the current site is the coordinator, it should shnply abort the transaction. If the current site is a subordinate, and it has not yet responded to the coordinator's *prepare* message, it can (and should) abort the transaction. If it is a subordinate and has voted *yes*, then it cannot unilaterally abort the transaction, and it cannot cOIUlnit either; it is blocked. It lnust periodically contact the coordinator until it receives a reply.

Failures of COffilnunication links are seen by active sites as failure of other sites that they are comnlunicating with, and therefore the solutions just outlined apply to this case as \-vell.

22.14.3 Two-Phase Commit Revisited

Now that we examined how a site recovers frolll a failure, and saw the interaction between the 2PC protocol and the recovery process, it is instructive to consider how 2PC can be refined further. In doing so, we arrive at a more efficient version of 2PC, but equally irnportant perhaps, we understand the role of the various steps of 2PC ruore clearly. Consider three basic observations:

1. l'he *ack* rnessages in 2PC are used to detennine when a coordinator (or the recovery process at a coordinator site following a crash) can 'forget' about a transaction *T*. lJntil the coordinator knows that all subordinates are aware of the cornrnit or abort decision for *T*, it IIlust keep inforrnation about *T* in the transaction table.

2. If the coordinator site fails after sending out *prepare* rnessages but before writing a cornrnit or abort log record, when it cornes back up, it has no inf'orruatiol1 abollt the transaction's connnit status prior to the crash. However, it is still free to abort the transaction unilaterally (beca,use it has not \vrittcn a conunit record, it can still cast a *no* vote itself). If another site inquires about the status of the transaction, the recovery process, as we have seen, responds \vith an *abort* rnessage. Therefore, in the absence of inforrnation, a transaction is *presumed to h..ave aborted.*

3. If a subtransaction does no updates, it has no changes to either redo or undo: in other words. its cornrnit or abort status is irrelevant.

The first two ol)servations suggest several refinements:

- When a coordinator aborts a transaction *T*, it can undo *T* and rerl10ve it fronl the transaction table irnrrlediately. After all\ rernoving *T* frorn the table results in a 'no infornatioI1' state with respect to *T*, and the default response (to an enquiry about *T)* in this state, \vhich is *abort*, is the correct response for an aborted transaction.

- By the same token, if a subordinate receives an *abort* Inessagc, it need not send an *ack* lnessage. rrhe coordinator is not waiting to hear frorn subordinates after sending an *abor't* 111essage! If, for SOlne reason, a subordinate that receives a *prepare* message (and voted *yes)* does not receive an *abort* or *commit* Inessage for a specified tirne-out interval, it contacts the coordinator again. If the coordinator decided to abort, there Inay no longer be an entry in the transaction table for this transaction, but the subordinate receives the default *abort* nlessage, whicll is the correct response.

- Because the coordinator is not waiting to hear froul subordinates after deciding to abort a transaction, the names of subordinates need not be recorded in the abort log record for the coordinator.

- All abort log records (for the coordinator as well as subordinates) can simply be appended to the log tail, instead of doing a force-write. After all, if they are not written to stable storage before a crash, the default decision is to abort the transaction.

The third basic observation suggests SOlne additional refinements:

- If a subtransaction does no updates (which can be easily detected by keeping a count of update log records), the subordinate can respond to a *prepare* 111essage from the coordinator with a *reader* message, instead of *yes* or *no*. The subordinate writes no log records in this case.

- When a coordinator receives a *reader* lnessage, it treats the Inessage as a *yes* vote, but with the optiInization that it does not send any lnore messages to the subordinate, because the subordinate's cornlnit or abort status is irrelevant.

- If all subtransactions, including the sllbtransaction at the coordinator site, send a *reader* luessagc, we do not need the second phase of the conunit protocol. Indeed, we can sirnply rernove the transaction frolH the transaction table, \vithout \vriting any log records at any site for this transaction.

The T'wo-Phasc Cornrnit protocol with the refinernents discussed in this section is called Two-Phase Commit with Presurned Abort.

22.14.4 Three-Phase Commit

A cornlnitprotocol called Three-Phase Conlrnit (3PC) can avoid blocking even if the coordinator site fails during recovery. The basic idea is that, when the coordinator sends out *prepare* rnessages and receives *yes* votes £1'0111 all subordinates, it sends all sites a *precommit* message, rather than a *cornrnit* rnessage. When a sufficient l1Ulllber-...-more than the lIlaxinlulll nUlnber of failures that nlust be handled········o*acks* have been received, the coordinator force-writes a *cornrn·it* log record and sends a *cornmit* lnessage to all subordinates. In 3PC, the coordinator effectively postpones the decision to cornrnit until it is sure that enough sites know about the decision to corn111it; if the coordinator subsequently fails, these sites can C0l11111Unicate with each other and detect that the transaction rnust be corllrnitted-conversely, aborted, if none of thern has received a *precomrnit* rnessage-'-without waiting for the coordinator to recover.

rrhe 3PC protocol ilnposes a significant additional cost during normal execution and requires that COlnrIlunication link failures do not lead to a network partition (wherein sorne sites cannot reach some other sites through any path) to ensure freedo111 fronl blocking. For these reasons, it is not used in practice.

22.15 REVIEW QUESTIONS

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

- Discuss the different rnotivations behind parallel and distributed databases. (Section 22.1)
- Describe the three lllain architectures for parallel DBMSs. Explain why the *shared-memory* and *shared-disk* approaches suffer frOlll *interference*. What can you say about the *speed-up* and *scale-up* of the *shared-nothing* architecture? (Section 22.2)
- Describe and differentiate *pipelined parallelism* and *data-partitioned parallelism.* (Section 22.3)
- Discuss the following techniques for partitioning data: *round-Tobin, hash,* and *range.* (Section 22.3.1)
- Explain how existing code can be parallelized by introducing *split* and *merge* operators. (Section 22.3.2)
- Discuss how each of the following operators can be parallized using data partitioning: *scanning, sorting, join.* Cornparc the use of sorting versus hashing for partitioning. (Section 22.4)
- What do we need to consider in optilllizing queries for parallel execution? Discuss interoperation parallelislll, left-deep trees versus bushy trees, and cost estimation. (Section 22.5)

- Define the tenns *distributed data independence* and *distributed transaction atomicity.* Are these concepts sllpported in current eornrnercial systerns? \Why not? What is the difference between *homogeneous* and *heterogeneous* distributed databases? (Section 22.6)

- Describe the three lllain architectures for distributed DBMSs. (Section 22.7)

  1\ relation can be distributed by *fragmenting* it or *replicating* it across several sites. Explain these concepts and ho\v they differ. Also, distinguish between *horizontal* and *vertical* fragrnentation. (Section 22.8)

- If a relation is fraglnented and replicated, each partition needs a globally unique nalne called the *Tclat'ion narnc.* Explain how such global naInes are created and the Inotivation behind the described approach to narning. (Section 22.9.1)

- Explain how rnetadata about such distributed data is rnaintained in a *dis-tr'ibuted catalog.* (Section 22.9.2)

- Describe a nauling scherne that supports distributed data independence. (Section 22.9.3)

- When processing queries in a distributed DBMS, the location of partitions of the relation needs to be taken into account. Discuss the alternatives when joining two two relations that reside on different sites. In particular, explain and describe the rnotivation behind the *Sernijoin* and *Bloornjoin* techniques. (Section 22.10.2)

- What issues rnust be considered in optirnizing queries over distributed data, in addition to where the data is located? (Section 22.10.3)

- What is the difference bet\veen *synchronous asynchronous* replication? Why has asynchronous replication gained in popularity? (Section 22.11)

- Describe the *'ooting* and *Tead-a'ny write-all* approaches to synchronous replication. (Section 22.11.1)

- Surnruarize the *peer-to-peer* and *primary site* approaches to asynchronolls replication. (Section 22.11.2)

- In prirnary site replication, changes to the prirnary copy Inust be propagated to secondary copies. What is done in the *Capture* and *Apply* steps? Describe *log-based* and *procedural* approaches to Capture and cornpare theln. What are the variations in scheduling the Apply step? Illustrate the use of asynchronolls replication in a data warehouse. (Section 22.11.2)

- What is a *subtransaction*? (Section 22.12)

- What are the choices for rnanaglng locks in a distributed DBMS? (Section 22.13)
- Discuss deadlock detection in a distributed database. Contrast the *centralized*, *hierarchical*, and *time-out* approaches. (Section 22.13.1)
- Why is recovery in a distributed DBMS rIlore cornplicated tha.n III a centralized systern? (Section 22.14)
- What is a *commit protocol* and \vhy is it required in a distributed database? Describe and compare *Two-Phase* and *Three-Phase Commit*. What is *blocking*, and how does the Three-Phase protocol prevent it? Why is it nonetheless not used in practice? (Section 22.14)

EXERCISES

Exercise 22.1 Give brief answers to the following questions:

1. What are the siruilarities and differences between parallel and distributed database rnan-agement systerns?
2. Would you expect to see a parallel database built using a wide-area network? \Vould you expect to see a distributed database built using a wide-area network? Explain.
3. Define the terms *scale-up* and *speed-up*.
4. Why is a shared-nothing architecture attractive for parallel database systerns?
5. The idea of building specialized hardware to run parallel database applications received considerable a tion but has fallen out of favor. Cornrnent on this trend.
6. What are th, antages of a distributed D131\1[8 over a centralized DBMS?
7. Briefly descr and cornpare the Client-Server and Collaborating Servers architectures.
8. In the Colla ting Servers architecture, \vhen a transaction is subrnitted to the DBMS, briefly des how its activities at various sites are coordinated. In particular, describe the role o .saction managers at the different sites, the concept of *subtransactions*, and the, .cept of *distributed transaction atomicity*.

Exercise 22.2 Give brief a,nswers to the follmving questions:

1. I)efine the tenus *fragmentation* and *replication* in tenns of where data is stored.
2. Wha is the difference between *synchronous* and *asynchronous* replication?
3. Define the tern1 *distributed data independence*. What does this Inean'with respect to quer:ying and updating data in the presence of data fragrnentation and replication?
4. Consider the *voting* and *read-any write-all* techniques for irnplementing synchronous replication. What are their respective pros and cons?
5. Give an overview of how asynchronous replication can be irnplemented. In particular, explain the terms *Capture* and *Apply*.
6. What is the difference between log-based and procedural irnplernentatiOlls of capture?
7. Why is giving database objects unique names rnore cennplicated in a distributed DBMS?

8. Describe a catalog organization that pennits any replica (of an entire relation or a frag- rnent) to be given a unique nam,e and provides the nanling infrastructure required for ensuring distributed data independence.
9. If infonuation from renlote catalogs is cached at other sites, what happens if the cached infoflllation becOInes outdated? How can this condition be detected and resolved?

Exercise 22.3 Consider a parallel DBMS in \vhich each relation is stored by horizontally partitioning its tuples across all disks:

Ernployees(eid: integer, *did:* integer, *sal:* real)
Departments(*did:* integer, *mgrid:* integer, *budget:* integer)

The *mgrid* field of DepartInents is the *e'id* of the manager. Each relation contains 20-byte tuples, and the *sal* and *budget* fields both contain unifonnly distributed values in the range O to 1 rnillion. The Enlployees relation contains 100,000 pages, the Departrnents relation contains 5,000 pages, and each processor has 100 buffer pages of 4,000 bytes each. The cost of one page I/O is t_d, and the cost of shipping one page is t_s; tuples are shipped in units of one page by waiting for a page to be filled before sending a rnessage frmn processor i to processor j. 'There are no indexes, and all joins that are local to a processor are carried out using a sort-rnerge join. Assurne that the relations are initially partitioned using a round-robin algorithlll and that there are 10 processors.

For each of the following queries, describe the evaluation plan briefly and give its cost in tenns of t_d and t_s. You should cornpute the total cost across all sites as well as the 'elapsed time' cost (i.e., if several operations are carried out concurrently, the tirne taken is the rnaxilnurn over these operations).

1. Find the highest paid ernployee.
2. Find the highest paid employee in the departrnent with *did* 55.
3. Find the highest paid ernployee over all departnHmts with *budget* less than 100,000.
4. Find the highest paid enlployee over all departlnents with *budget* less than 300,000.
5. Find the a;verage salary over all departments with *budget* less than 300,000.
6. Find the salaries of all rnanagers.
7. Find the salaries of all rnanagers who manage a departrnent with a budget less than 300,000 and eaTll rnore than 100,000.
8. Print the *eids* of all elnployees, ordered by increasing salaries. Each processor is connected to a separate printer, and the answer can appear as several sorted lists, each printer] by a different processor, as long as we can ol)tain a fully sorted list by concatenating the printed lists (in sorne order).

Exercise 22.4 Consider the saIne scenario as in Exercise 22.3, except that the relations are originally partitioned using range partitionirlg on the *sal* and *budget* fields.

Exercise 22.5 Repeat Exercises 22.3 and 22.4 with (i) 1 processor, and (ii) lelO processors.

Exercise 22.6 COllsicler the Ernployees and Departments relations described in Exercise 22.3. They are now stored in a distributed DBMS with all of Employees stored at Naples and all of Departlnents stored at Berlin. There arc no indexes on these relations. The cost of various operations is as describecl in Exercise 22.:3. Consider the query:

```
SELECT  *
FROM    EInployees E, Dcpartrncnts D
WHERE   E.eid = I).Ingrid
```

'The query is posed at Delhi, and you are told that only 1 percent of ernployees are IIlanagers. Find the cost of answering this query using each of the following plans:

1. Ship Departrnents to Naples, cornpute the query at Naples, then ship the result to Delhi.
2. Ship Ernployees to Berlin, cornpute the query at Berlin, then ship the result to Delhi.
3. COInpute the query at Delhi by shipping both relations to Delhi.
4. COlnpute the query at Naples using BloOlnjoin; then ship the result to Delhi.
5. Compute the query at Berlin using Bloornjoin; then ship the result to Delhi.
6. Cornpute the query at Naples using Sernijoin; then ship the result to Delhi.
7. COInpute the query at Berlin using Sernijoin; then ship the result to Delhi.

Exercise 22.7 Consider your answers in Exercise 22.6. Which plan ll1inin1izes shipping costs? Is it necessarily the cheapest plan? Which do you expect to be the cheapest?

Exercise 22.8 Consider the Ernployees and Departments relations described in Exercise 22.3. They are now stored in a distributed DBMS with 10 sites. The DepartInents tuples are horizontally partitioned across the 10 sites by *did,* with the same nUInber of tuples assigned to each site and no particular order to how tuples are assigned to sites. The Employees tuples are sirnilarly partitioned, by *sal* ranges, with $sal \leq$ 100,000 assigned to the first site, 100,000 $<$ $sal \leq$ 200,000 assigned to the second site, and so Oll. In addition, the partition $sal \leq$ 100,000 is frequently accessed and infrequently updated, and it is therefore replicated at every site. No other EU1ployees partition is replicated.

1. Describe the best plan (unless a plan is specified) and give its cost:
   (a) Cornpute the natural join of Enlployees and Departlnents by shipping all fragrnents of the slImller relation to every site containing tuples of the larger relation.
   (b) Find the highest paid ernployee.
   (c) Find the highest paid clnployee with salary less than 100,000.
   (d) Find the highest paid ernployee with salary between 400,000 and 500,000.
   (e) 'Find the highest paid clnployee with salary between 450,000 and 550,000.
   (f) Find the highest paid rnanager for those departnwnts stored at the query site.
   (g) Find the highest pajd lnanager.
2. ASSU111ing the sarne data distribution, describe the sites visited and the locks obtained for the foll()\ving update transactions, assuming that *synchronous* replication is used for the replication of Employees tuples with $sal \leq$ *100,* ()(){}:
   (a) Give employees with salary less than 100,000 a 10 percent raise, with a Inaxirnurn salary of 100,000 (i.e., the raise cannot increase the salary to rnore than 100,(00).
   (b) Give all ernployees a 10 percent raise. The conditions of the original partitioning of Elnployees IIlust still be satisfied after the update.
3. AssuIning the saIne data distribution, describe the sites visited and the locks obtained for the following update transactions, assuming that *asynchronous* replication is used for the replication of Ernployees tuples with $sal \leq$ 100,000.

For all employees with salary less than 100,000 give them a 10 percent raise, with a maximum salary of 100,000.

Give all employees a 10 percent raise. After the update is completed, the conditions of the original partitioning of Eluployecs rnust still be satisfied.

Exercise 22.9 Consider the Elnployees and Departments tahles from Exercise 22.:3. You are a DBA and you need to decide how to distribute these two tables across t\vo sites, IVlanila and Nairobi. Your DBMS supports only unclustered 13+ tree indexes. You have a choice between synchronous and asynchronous replication. For each of the following scenarios, describe how you would distribute thenl and what indexes you would build at each site. If you feel that you have insufficient information to make a decision, explain briefly.

1. Half the departInents are located in Manila and the other half are in Nairobi. Departrnent information, including that for ernployees in the depart1nent, is changed only at the site where the departrnent is located, but such changes are quite frequent. (Although the location of a depart1nent is not included in the Departrnents schclna, this inforrnation can be obtained frorn another table.)
2. Half the departrnents are located in Manila and the other half are in Nairobi. Departrnent information, including that for errlployees in the departrnent, is changed only at the site where the departrnent is located, but such changes are infrequent. F'inding the average salary for each departrnent is a frequently asked query.
3. Half the departlnents are located in Ivlanila and the other half are in Nairobi. Ernployees tuples are frequently changed (only) at the site where the corresponding departrrlent is located, but the Depart1nents relation is aJulOst never changed. Finding a given ernployee's rnanager is a frequently asked query.
4. Half the e1nployees work in Manila and the other half work in Nairobi. E1nployees tuples are frequently changed (only) at the site where they work.

Exercise 22.10 Suppose that the Ernployees relation is stored in l\1adison and the tuples with $sal \leq 1.00{,}000$ are replicated at New York. Consider the following three options for lock rnanagernent: all locks managed at a *single site*, say, 1VIilwaukee; *primary copy* with l'vladison being the primary for Employees; and *fully distributed*. For each of the lock rnanagernent options, explain what locks are set (and at which site) for the following queries. Also state frorn which site the page is reacl.

1. A query at Austin wants to read a page of Erllployees tuples \vith $sal \leq$ 50,()OO.
2. A query at Madison wants to read a page of Employees tuples with $sal \leq 50{,}000$.
3. A query at New York wants to read a page of Employees tuples "vith $sal \leq$ *50,000.*

Exercise 22.11 Briefly answer the following questions:

1. Compare the relative merits of centralized and hierarchical deadlock detection in a distril)lIted DBMS.
2. What is a *phantom deadlock*? Give an example.
3. Give an example of a distributed DBMS \vith three sites such that no two local waits-for graphs reveal a deadlock, yet there is a global deadlock.
4. Consider the following rnoclification to a local waits-for graph: Add a new node T_{ext}, and for every transaction T_i that is waiting for a lock at another site, add the edge T_i T_{ext}. A.lso add an edge $T_{ext} \rightarrow T_i$ if a transaction executing at another site is waiting for T_i to release a lock at this site.

If there is a cycle in the modified local waits-for graph that does not involve T_{ext}, what can you conclude? If every cycle involves T_{ext}, what can you conclude?

Suppose that every site is assigned a unique integer Whenever the local waits-for graph suggests that there Blight be a global deadlock, send the local waits-for graph to the site with the next higher site-id. At that site, combine the received graph with the local waits-for grap,h. If this cornbined graph does not indicate a deadlock, ship it on to the next site, and so on, until either a cleadlock is detected or we are back at the site that originated this round of deadlock detection. Is this scheme guaranteed to find a global deadlock if one exists?

Exercise 22.12 Tirnestarnp-based concurrency control schernes can be used in a distributed DBMS, but we rllust be able to generate globally unique, rllonotonicaJly increasing timestamps without a bias in favor of anyone site. One approach is to a,ssign timestamps at a single site. Another is to use the local clock tiTne and to append the site-iei. A third scherne is to use a counter at each site. COIllpare these three approaches.

Exercise 22.13 Consieler the rIlultiple-granlllarity locking protocol described in Chapter 18. In a distributed DBMS, the site containing the root object in the hierarchy can becmne a, bottleneck. You hire a database consultant who tells you to rnodify your protocol to allow only intention locks on the root and irnplicitly grant all possible intention locks to every transaction.

1. Explain why this rnodification \vorks correctly, in that transactions continue to be able to set locks on desired parts of the hierarchy.
2. Explain how it reduces the demand on the root.
3. Why is this idea not included as part of the standard rllultiple-granularity locking protocol for a centralized DBMS?

Exercise 22.14 Briefly answer the following questions:

1. Explain the need for a cornmit protocol in a distributed DBMS.
2. Describe 2PC. Be sure to explain the need for force-writes.
3. Why are *ack* messages required in 2PC?
4. What are the differences between 2PC and 2PC with PresulTled Abort?
5. Give an exarnple execution sequence such that 2PC and 2PC 'with Presurned Abort: generate an identical sequence of actions.
6. Give an exarIlple execution sequence such that 2PC and 2PC with Presumed Abort generate different sequences of actions.
7. \Vhat is the intuition behind 3PC? What are its pros and cons relative to 2PC?
8. Suppose that a site gets no response frorn another site for a long time. Can the first site tell whether the connecting link has failed or the other site has failed? How is such a failure handled?
9. Suppose that the coordinator inclucles a list of aU subordinates in the *prepare* message. If the coordinator fails after sending out either an *abort* or *commit* message, call you suggest a way for active sites to terrninate this tra.nsaction without wajting for the coordinator to recover? Assume that some but not all of the *abort* or *commit* messages frOln the cocn'clinator are lost.

III Suppose that 2PC with PresuIl1ed Abort is used as the cOInmit protocol. Explain how the systmll recovers froIn failure and deals with a particular transaction T in each of the following cases:

(a) A subordinate site for T fails before receiving a *prepare* rnessage.

(b) A subordinate site for T fails after receiving a *prepare* rnessage but before rnaking a decision.

(c) A subordinate site for T fails after receiving a *prepare* lnessage and force-writing an abort log record but before responding to the *prepare* message.

(d) A subordinate site for T fails after receiving a *prepare* message and force-writing a prepare log record but before responding to the *prepare* lnessage.

(e) A subordinate site for T fails after receiving a *prepare* rnessage, force-writing an abort log record, and sending a *no* vote.

(f) The coordinator site for T fails before sending a *prepare* lnessage.

(g) The coordinator site for T fails after sending a *prepare* lllCssage but before collecting all votes.

(h) The coordinator site for T fails after writing an *abort* log record but before sending any further rnessages to its subordinates.

(i) The coordinator site for T fails after writing a *comrnit* log record but before sending any further rnessages to its subordinates.

(j) The coordinator site for T fails after writing an *end* log record. Is it possible for the recovery process to receive an inquiry about the status of T frolll a subordinate?

Exercise 22.15 Consider a heterogeneous distributed DBMS.

1. Define the terms *multidatabase system* and *gateway.*
2. Describe how queries that span multiple sites are executed in a rnultidatabase systern. Explain the role of the gateway with respect to catalog interfaces, query optirnizatiOll, and query execution.
3. Describe how transactions that update data at rnultiple sites are executed in a lllulti-database systern. Explain the role of the gateway with respect to lock rnanagernent, distributed deadlock detection, Two-Phase COllnnit, and recovery.
4. SChell1aS at different sites in a llnI1tidatabase systern are probably designed independently. T'his situation can lead to *semantic heterogeneity;* that is, units of rneasure rnay differ across sites (e.g., inches versus centirneters), relatiolls containing essentially the same kind of infonnation (e.g., employee salaries and ages) rnay have slightly different schernas, and so on. What ilnpact does this heterogeneity have on the end user? In particular, COllunent on the concept of distributed data independence in such a systcrIl.

BIBLIOGRAPHIC NOTES

\Vork on parallel algorithrns for sorting and various relational operations is discussed in the bibliographies for Chapters 13 and 14. Our discussion of parallel joins follows [220], and our discussion of parallel sorting follows [223]. DeWitt and Gray make the case that for future high performance database systeuls, parallelisnl will be the key [221]. Scheduling ill parallel database systenls is discusserl in [522). [496] contains a good collection of papers on query processing in parallel database systems.

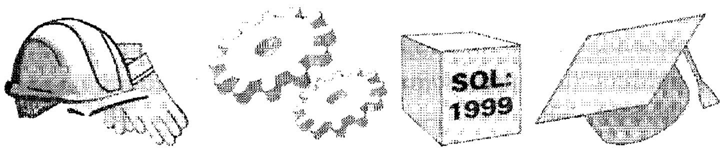
Textbook discussions of distributed databases include [78, 144, 580]. Good survey articles include [85], which focuses on concurrency control; [637], which is about distributed databases in general; and [785], which concentrates on distributed query processing. Two InajaI' projects in the area were 8DD-1 [636] and R* [777]. Fragmentation in distributed databases is considered in [157, 207]. Replication is considered in [11, 14, 137,239,238, 388, 385, 335, 552, 600]. For good overviews of current trends in asynchronous replicatioll, see [234, 709, 772]. Papers on view maintenance mentioned in the bibliographic notes of Chapter 21 are also relevant in this context. Olston considers techniques for trading of performance versus precision in a replicated environment [571, 572, 573].

Query processing in the 8DD-1 distributed database is described in [88]. One of the notable aspects of 8DD-1 query processing was the extensive use of 8emijoins. Theoretical studies of Semijoins are presented in [83, 86, 414]. Query processing in R* is described in [667]. The R* query optimizer is validated in [500]; much of our discussion of distributed query processing is drawn from the results reported in this paper. Query processing in Distributed Ingres is described in [247]. Optimization of queries for parallel execution is discussed in [297,323,383]. Franklin, Jonsson, and Kossman discuss the trade-offs between *query shipping,* the more traditional approach in relational databases, and *data shipping,* which consists of shipping data to the client for processing and is widely used in object-oriented systerns [284]. A good recent survey of distributed query processing techniques can be found in [450].

Concurrency control in the 8DD-1 distributed database is described in [91]. Transaction management in R* is described in [547]. Concurrency control in Distributed Ingres is described in [714]. [740] provides an introduction to distributed transaction rrlanagement and various notions of distributed data independence. Optimizations for read-only transactions are discussed in [306]. Multiversion concurrency control algorithrlls based on timestamps were proposed in [620]. Timestamp-based concurrency control is discussed in [84, 356]. Concurrency control algorithms based on voting are discussed in [303, 318, 408, 452, 732]. The rotating prirnary copy scheme is described in [538]. Optimistic concurrency control in distributed databases is discussed in [660], and adaptive concurrency control is discussed in [488].

Two-Phase Commit was introduced in [466, 331]. 2PC with Presumed Abort is described in [546], along with an alternative called *2PC with Presum.ed Cornmit.* A variation of Presumed Cornrrlit is proposed in [465]. Three-Phase COlnrrlit is described in [692]. The deadlock detection algorithnls in R* are described in [567]. Many papers discuss deadlocks, for exaInple, [156, 243, 526, 632]. [441] is a survey of several algoritluns in this area. Distributed clock synchronization is discussed by [464]. [333] argues that distributed data independence is not always a good idea, clue to processing and adlninistrative overheads. The ARIES algorithrl1 is applicable for distributed recovery, but the details of how rnessages should be handled are not discussecl in [544]. The approach taken to recovery in SDD-1 is described in [43]. [114] also addresses distributed recovery. [444] is a survey article that discusses concurrency control and recovery in distributed systerIls. [95) contains several articles on these topics.

IVlultidatabase systems are discussed in [10, 113, 230, 231, 242,476, 485, 519, 520, 599, 641, 765, 797]; sec [112, 486, 684] for surveys.



23

OBJECT-DATABASE SYSTEMS

- What are object-database systerlls and what new features do they support?
- What kinds of applications do they benefit?
- \Vhat kinds of data types can users de.fine?
- What are abstract data types and their benefits?
- What is type inheritance and why is it useful?
- What is the irnpact of introducing object ids in a database?
- How can we utilize the new features in database design?
- What are the new implelncntation challenges?
- \Vhat difFerentiates object-relational and object-oriented DBMSs?
- Key concepts: user-defined data types, structured types, collection types; data abstraction, rnethocls, encapsulation; inheritance, early and late binding of rnethods, collection hierarchies; object identity, reference types, shallow and deep equality

with Joseph M. HeHerstein
University of California-Berkeley

You know Iny Inethods, Watson. Apply theIn.

Arthur Conan Doyle, *The Memoirs of Sherlock Holmes*

Relational database systeros support a sruaU, fixed collection of data types (e.g., integers, dates, strings),\vhich has proven adequate for traditional application domains such as adruinistrative data processing. In many application dornains, however, rnuch 1nore complex kinds of data Blust be handled. Typically this cornplex data has been stored in OS file systems or specialized data structures, rather than in a DBMS. Examples of dornains with cOJ.uplex data include cornputer-aided design and rnodeling (CA.D/CAM), multimedia repositories, and docurnent Hl8.Jlagernent.

As the arnount of data grows, the luany features offered by a DBIvISfor exarIl-pIe, reduced application developnlent time, concurrency control and recovery, indexing support, and query capabilities······becoruéincreasingly attractive and, ultimately, necessary. To support such applications, a DBMS HUlst support cornplex data types. ()bject-oriented concepts strongly influenced efforts to enhance database support for cornplex data and led to the developrnent of object-database systelus, \vhich we discuss in this chapter.

Object-database systerlls have developed along two distinct paths:

- Object-Oriented Database Systems: Object-oriented database systerns are proposed as an alternative to relational systerlls and are ainled at application dornains where cODlplex objects playa centra,} role. The approach is heavily influenced by object-oriented prograrllrlling languages and can be understood as an atternpt to add DBMS functionality to a prograunning language environrnent. The ()bject Database :M:anagenlcnt Group (()DMG) has developed a standard Object Data Model (ODM) and Object Query Language (OQL), which are the equivalent of the SQL standard for relational database systerns.
- Object-Relational])atabase Systenls: ()bject-relational database s.ysterns ca,n be thought of as an atternpt to extend relational database systerns with the functionality necessary to support a broader class of applications and, in many ways, provide a bridge between the relational and object-oriented paTadiguls. The SQL:1999 standard extends SQL to incorporate support for the ol)ject-relationaJ rnode1 of data.

We use acronyms for relational, object-oriented, and object-relational database rnanagernent systems (RDBMS, OODBMS, ORJDBMS). In this chapter, we focus 011 ORDBMSs and ernphasize how they can be viewed as a developrnent of HJ)BMSs, rather than as an entirely different paradigrn, as exernplified l)y the evolution of SQL:1999.

We concentrate on developing the fUlldarnental concepts rather than presenting SQL:1999; some of the features we discuss are not inc.luded in SQL:1999.

Nonetheless, we have chosen to emphasize concepts relevant to SQL:1999 and its likely future extensions. We also try to be consistent with SQL:1999 for notation, although we occasionally diverge slightly for clarity. It is hnportant to recognize that the rnain concepts discussed are COIllITIOn to both ORDBMSs and ()()DBNISs; we discuss how they are supported in the ODLjOQL standard proposed for OODBMSs in Section 23.9.

RDB1\JIS vendors, including IBM, Inforrnix, and Oracle, are adding ORDBMS functionality (to varying degrees) in their products, and it is inlportant to recognize how the existing body of knowledge about the design and inlple- rnentation of relational databases can be leveraged to deal with the ORDBMS extensions. It is also ilnportant to understand the challenges and opportunities these extensions present to database users, designers, and irnplernentors.

In this chapter, Sections 23.1 through 23.6 introduce object-oriented concepts. The concepts discussed in these sections are COlunlon to both OODBMSs and ORDBJVISs. We begin by presenting an example in Section 23.1 that illustrates why extensions to the relational rnodel are needed to cope with some new application dornains. 'This is used as a running exarnple throughout the chapter. We discuss the use of type constructors to support user-defined structured data types in Section 23.2. We consider what operations are supported on these new types of data in Section 23.3. Next, we discuss data encapsulation and abstract data types in Section 23.4. We cover inheritance and related issues, such as rnethod binding and collection hierarchies, in Section 23.5. We then consider objects and object identity in Section 23.6.

We consider how to take advantage of the new object-oriented concepts to do OI{DBMS database design in Section 23.7. In Section 23.8, we discuss SOHle of the new irnplernentation challenges posed by object-relational systerns. We discuss ()I)L and OQL, the standards for OODBMSs, in Section 23.9, and then present a brief cornparison of ()R,DBMSs and OODBMSs in Section 23.10.

23.1 MOTIVATING EXAMPLE

As a specific exarnple of the need for object-relational systcrlls, we focus on a new business data processing probler.n that is both harder and (in our view) rnorc entertaining than the dollars and cents bookkeeping of previous decades. Today, cornpanies in industries such as entertainruent are in the business of selling *bits;* their basic corporate assets are not tangible products, but rather software artifacts such as video (l,nd audio.

We consider the fictional Dinky Entertaiurnent Corupa,ny, a large IIollywood conglornerate whose main assets are a collection of cartoon characters, especially the cuddly and internationally beloved IIerbert the \VarIll. Dinky has several IIerbert the \Vornlfihns, rnany of which are shown in theaters around the world at any given time. Dinky also rnakes a good deal of rnoney licensing Herbert's irnage, voice, and video footage for various purposes: action figures, video gaInes, product endOrSelllents, and so on. I)inky's database is used to lnanage the sales and leasing records for the various IIerbert-related products, as well as the video and audio data that rnake up IIerbert's lllany filIns.

23.1.1 New **Data** Types

The basic problern confronting Dinky's database designers is that they need support for considerably richer data types than is available in a relational DBMS:

- User-defined data types: Dinky's assets include Herbert's iIllage, voice, and video footage, and these rnust be stored in the database. To handle these new types, we need to be able to represent richer structure. (See Section 23.2.) Further, we need special functions to rnanipulate these objects. For example, we may want to write functions that produce a cOlnpressed version of an irnage or a lower-resolution image. By hiding the details of the data structure through the functions that capture the behavior, we achieve *data abstract'ion,* leading to cleaner code design. (See Section 23.4.)

- Inheritance: As the nurnber of data types grows, it is irnportant to take advantage of the cornrnonality between different types. :For exarnple, both cOInpressed irnages and lower-resolution irnages are, at SOlne level, just ilnages. It is therefore desirable to *inherit* some features of iluage objects while defining (and later Inanipulating) cOInpressed irnage objects and lower-resolution irnage objects. (See Section 23.5.)

- Object Identity: Given that seHne of the new data types contain very large instances (e.g., videos), it is iInportant not to store copies of objects; instead, we IIIUSt store *references*, or *pointers*, to such objects. In turn, this underscores the need for giving objects a unique *object identity*, which can be used to refer or 'point' to theln frorn elsewhere in the data. (See Section 23.6.)

Flow lnight we address these issues in an RDBMS? We could store ilnages, videos, and so on as BLC)Bs in current relational systems. A binary large object (BLOB) is just a long stream of bytes, and the DBMS's support consists of storing and retrieving BLC)Bs in such a rnanner that a user does not have to worry about the size of the BLC)B; a 13LC}B can span several pages, unlike a traditional attribute. All further processing of the BLC)B has to be done by the user's application progranl, in the host language in \vhich the

> **The SQL/MM Standard:** SQL/MM is an eillerging standard that builds upon SQL:1999's new data types to define extensions of SQL:1999 that facilitate handling of coruplex multimedia data types. SQL/MM is a rnultipart standard. Part 1, SQL/MM Framework, identifies the SQL:1999 concepts that are the foundation for SQL/MM extensions. Each of the relnaining parts addresses a specific type of ccnnplex data: Full Text, Spatial, Still Image, and Data Mining. SQL/MM anticipates that these new coruplex types can be used in colurnns of tables as field values.

> **Large Objects:** SQL:1999 includes a new data type called LARGE OBJECT or LOB, with two variaJ1ts called BLOB (binary large object) and CLOB (character large object). This standardizes the large object support found in lnany current relational DBMSs. LOBs cannot be included in priruary keys, GROUP BY, or ORDER BY clauses. rrhey can be cornpared llsing equa.lity, inequality, and substring operations. A LOB has a locator that is essentially a unique id and allows LOBs to be rnanipulated without extensIve copYIng.
> L()Bs are typically stored separately froIn the data records in whose fields they appear. IBM DB2, InforInix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support LOBs.

SQL code is ernbedded. This solution is not efficient because we are forced to retrieve all BLOBs in a collection even if rnost of the111 could be filtered out of the answer by applying user-defined functions (within the DBMS). It is not satisfactory frorn a data consistency standpoint either, because the selnantics of the data now depends heavily on the host la,nguage application code and cannot be enforced by the DBMS.

As for structured types and inheritance, there is simply no support in the relational Dlodel. We are forced to map data 'with such cOlnplex structure into a collection of flat tables. (We saw examples of such rnappings when we discussed the translation frorH ER diagrarns with inheritance to relations in Chapter 2.)

This application clearly requires features not available in the relational Inode1. As an illustration of these features, Figure 2:3.1 presents SQL:1999 :DDL staternents for a l)Ortion of Dinky's ()HJ)I31VIS schema used in subsequent examples. Although the 1)})L is very sirnilar to that of a traditional relational systeru, some irnportant distinctions highlight the new data rnodeling capabilities of an ORDBMS. A quick glance at the l)DL staternents is sufficient for now; we study them in detail in the next section, after presenting s0111e of the basic concepts that our sanlple application suggests are needed in a next-generation DBMS.

1. CREATE TABLE Frames
   (*frameno* **integer**, *image* **jpeg_image**, *category* **integer**);
2. CREATE TABLE Categories
   (*cid* **integer**, *name* text, *lease_price* float, *comments* text);
3. CREATE TYPE theater_t AS
   ROW(*tno* integer, *name* **text**, *address* text, *phone* text)
   REF IS SYSTEM **GENERATED**;
4. CREATE TABLE Theaters OF theater_t REF is tid SYSTEM GENERATED;
5. CREATE TABLE Nowshowing
   (*film* integer, *theater* REF(theater_t) SCOPE rrheaters, *start* date,
   end date);
6. CREATE TABLE FilIns
   (*filrnno* integer, *title* text, *stars* VARCHAR(25) ARRAY [10]),
   director text, *budget* float);
7. CREATE TABLE Countries
   (*name* text, *boundary* polygon, *population* integer, *language* text);

Figure 23.1 SQL:1999 DDL Staternents for Dinky Schema

23.1.2 Manipulating the New Data

Thus far, we described the new kinds of data that rnust be stored in the Dinky database. We have not yet said anything about how to *use* these new types in queries, so let us study two queries that I)inky's database needs to support. The syntax of the queries is not critical; it is sufficient to understand what they express. We return to the specifics of the queries' syntax later.

()ur first challenge COIn8S frorn the Clog breakfast cereal cornpany. Clog produces a cereal called I)elirios and it wants to lease an irnage of Herbert the Worm in front of a sunrise to incorporate in the I)elirios box design. A query to present a collection of possible irnages and their lease prices can be expressed in SQL-like syntax as in Figure 2:3.2. I)inky has a nUJn1>er of methods written in an irnperative language like Java and registered with the database systern. These methods can be used in queries in the sallIe way as built-ill methods, such as =. , −, <, >, are used in a relational language like SQL. The *thumbnail* IJlethod in the Select clause produces a srnaU version of its full-size input: image. The *is_sunrise* rnethod is a boolean function that analyzes an irnage and returns *true* if the image contains a sun.rise; the *is_herbert* Inethod returns *true* if the image contains a picture of l1erbert. rrhe query produces the frame code number, irnage thurnbnail, and price for all frames that contain Herbert and a sunrise.

```
SELECT F.fnuneno, thulnbnail(F.irnage), C.lease_price
FROM   Fralnes F, Categories C
WHERE  F.category = C.cid AND is.Bllnrise(F.irnage) AND isJlerbert(F.inlage)
```

Figure 23.2 Extended SQL to Find Pictures of Herbert at Sunrise

The second challenge carnes froIn Dinky's executives. They know that Delirios is exceedingly popular in the tiny country of A.ndorra, so they want to lIlake sure that a number of Herbert filIns are playing at theaters near Andorra when the cereal hits the shelves. To check on the current state of affairs, the executives want to find the 11aInes of all theaters showing Herbert fihns within 100 kilorneters of Andorra. Figure 23.3 shows this query in an SQL-like syntax.

```
SELECT N.theater··-->na1ne, N.theater->address, F.title
FROM   Nowshowing N, Filrns F, Countries C
WHERE  N.film = F.filrnno AND
       overlaps(C.bollndary, radius(N.theater->address, 100)) AND
       C.name = 'Andorra' AND 'Herbert the Worm' = F.stars[l]
```

Figure 23.3 Extended SQL to Find Herbert Films Playing near Andorra

The *theater* attribute of the Nowshowing table is a reference to an object in another table, which has attributes *narne, address*, and *location.* This object referencing allows for the notation *N.theater-> narne* and *N.theater···>address,* each of which refers to attributes of the theater_t object referenced in the Nowshowing row *N*. The *stars* attribute of the *tUrns* table is a set of narnes of each [ibn's stars. The *radius* nlethod returns a circle centered at its first argulllent with radius equal to its second argurnent. The overlaps rnethod tests for spatial overlap. Nowshowing and Filrns are joined by the equijoin clause, \vhile Nowshowing and Countries are joined by the spatial overlap clause. The selections to 'Andorra' and fiInls containing 'Herbert the vVorrn' cornplete the query.

rrhcse two object-relational queries are sirnilar to SQL-92 queries but have soinc unusual features:

- User-Defined Methods: User-defined abstract types are rnanipulated via their 1nethods, for exalnple, *is_herbert* (Section 23.2).
- Operators for Structured Types: A.long with the structured types available in the data rnodel, ()R,DBMSs provide the natural Inethods for those types. For exarnple, the ARRAY type supports the standard array

operation of accessing an array elenlent by specifying the index; *F.stars[l]* returns the first elernent of the array in the *staTs* cohllnn of film *F* (Section 23.3).

- **Operators for Reference Types:** Reference types are *dereferenced* via. an arrow (–> notation (Section 23.6.2).

To suuullarize the points highlighted by our 1110tivating exanlple, traditional relational systenls offer liInited flexibility in the data types available. Data is stored in tables and the type of each field value is lirnited to a siulple atornic type (e.g., integer or string), with a sl11all, fixed set of such types to choose frarn. This lirnited type systern can be extended in three Inain ways: user-defined abstract data types, structured types, and reference types. Collectively, we refer to these new types as **complex types**. In the rest of this chapter, we consider how a DBMS can be extended to provide support for defining new complex types and rnanipulating objects of these new types.

23.2 STRUCTURED DATA TYPES

SQL:1999 allows users to define new data types, in addition to the built-in types (e.g., integers). In Section 5.7.2, we discussed the definition of new *distinct* types. Distinct types stay within the standard relational model, since values of these types rnust be atornic.

SQL:1999 also introduced two **type constructors** that allow us to define new types with internaJ structure. Types defined using type constructors are called **structured types**. This ta.kes us beyond the relational model, since field values need no longer be atornic:

- RDW(n1 t_1, ..., *nn* t_n): A type representing a row, or tuple, of *n* fields \vith fields *11,1*, ..., n_n of types $t_1, \ldots, t_n$ respectively.
- base ARRAY [iJ): A type representing an array of (up to) i base-type iterns.

The **theater_t** type in Figure 23.1 illustrates the new ROW data type. In SQL:1999, the ROW type has a special role because every table is a collection of ro\vs -.·every table is a set of 1'o\vs or a. rnultiset of rc)\vs. Values of other types can appear only as field values.

The *stars* field of table Filrns illustrates the new ARRAY type. It is an array of upto 10 elements, each of \vhich is of type VARCHAR(25). Note that 10 is the rnaxirnurn nurnber of elements in the array; at any tiTne, the array (unlike, say,

> **SQL:1999 Structured Data Types:** Several conunercial systems, including IBM DB2, Infonnix UDS, and Oracle 9i support the ROWand ARRAY constructors. The `listof`, `bagof`, and setof type constructors are :not included in SQL:1999. Nonetheless, commercial systerIls support sorne of these constructors to varying degrees. Oracle supports nested relations and arrays, but does not support fully cornposing these constructors. InfOI'rnix supports the setof, hagof, and Ustof constructors and allows thern to be cornposed. Support in this area varies \videly across vendors.

in C) can contain fewer elenlcnts. Since SQL:1999 does not support rnultidi-Inensional arrays, *vector* rnight ha,ve been a rnore accura,te narne for the array constructor.

The power of type constructors cornes froIn the fact that they can be cornposed. The following row type conta,ins a field that is an array of at Inost 10 strings:

ROW(filrnno: integer, *stars:* VARCHAR(25) ARRAY [10])

The row type in SQL:1999 is quite general; its fields can be of any SQL:1999 data type. Unfortunately, the arra.y type is restricted; elernents of an array cannot be arrays thcrnselves. Therefore, the following definition is illegal:

(integer ARRAY [5]) ARRAY [10]

23.2.1 Collection Types

SQL:1999 supports only the ROW a,nel ARRAY type constructors. Other COUUllon type constructors include

- listof(base): A. type representing a sequence of base-type itcrllS.
- setof(base): A type representing a *set* of base-type HeIns. Sets cannot contain duplicate elements.
- bagof(base): A type representin.g a, *bag* or *multiset* of base-type iterns.

Types llsing listof, ARRAY, bagof, or setof as the outennost type constructor are sometimes referred to as collection types or bulk data types.

The lack of support for these collection types is recognized as a weakness of SQL:1999's support for cornplex objects and it is quite possible that SODle of these collection types will be added in future revisions of the SQL standard.[1]

23.3 OPERATIONS ON STRUCTURED DATA

rIhe I)131V18 provides built-in Inethods for the types defined using type constructors. These lnethods are analogous to built-in operations such as addition and rIlultiplication for atcnnic types such as integers. In this section we present the Illethods for various type constructors and illustrate ho\v SQL queries can create and rnanipulate values with structured types.

23.3.1 Operations on Rows

Given an iteul i whose type is ROW(n_1 t_1, ..., n_n t_n), the field extraction rnethod allo\vs us to access an individuaJ field n_k llsing the traditional clot notation $i.n_k$. If ro\v constructors are nested in a type definition, dots rnay be nested to access the fields of the nested row; for exarnple $i.n_k.m_l$. If we have a collection of rows, the dot notation gives us a collection as a result. :For exarnple, **if** i is a list of rows, $i.n_k$ gives us a list of itcrns of type t_n; if i is a set of rows, $i.n_k$ gives us a set of iterns of type t_n.

[This nested-dot notation is often called a path expression, because it describes a path through the nested structure.

23.3.2 ()peratiolls on Arrays

Array types support an 'array index' rnethod to allow 11sers to access array iterns at a, particular offset. A. postfix 'square bracket' syntax is usually used. Since the nuruber of elernents can vary, there is an operator (CARDINALITY) that returns the nU1nbe1' of elerIlents ııı the array. The varia,hle nurnl)er of elernents also rn.otivates an operator to C:Ollcatenate two arrays. The following exanlple illustrates these operations on SQL:1999 arrays.

```
SELECT  F.fillnIlo, (F.staTs ‖ ['Brando', 'Pacino'])
FROM    FilrnsF
WHERE   CARDINALITY(F.stars) < 3 AND F.stars[1]='Redford'
```

1According to Jinl Melton, the editor of the SQL:1999 standard, these collection types were considered for inclusion but omitted because some problems with their specifications were discovered too late for correction in the SQL:1999 time-frame.

For each fibn with Redford as the first star[2] and fewer than three stars, the result of the query contains the film's array of stars concatenated with the array containing the two elcrnents 'Brando' and 'Pacino'. Observe how a value of type array (containing Brando and Pacino) is constructed through the use of square brackets in the SELECT clause.

23.3.3 Operations on Other Collection Types

Although only arrays are supported in SQL:1999, future versions of SQL are expected to support other collection types, and we consider what operations are appropriate over these types of data. provide such operations. Our discussion is illustrative and not Ineant to be cOlnprehensive. For exarnple, one could additionally allow aggregate operators *count*, *sum*, *avg*, *max*, and *rnin* to be applied to any object of a collection type with an appropriate base type (e.g., INTEGER). ()ne could also support operators for type conversions. For exaInple, one could provide operators to convert a rnultiset object to a set object by elirninating duplicates.

Sets and Multisets

Set objects can be cornpared using the traditional set methods $\subset, \subseteq, =, \supseteq, \supset$. An iteln of type setof(faa) can be cornpared with an iteln of type faa using the $\in$ rnethod, as illustrated in Figure 23.3, which contains the cornparison *'Herbert the Worm'* $\in$ *F.stars*. T\vo set objects (having elernents of the saIne type) can be cornbined to forIlI a new object using the $\cup$, $\cap$, and $-$ operators.

Each of the Inethods for sets can be defined for Inultisets, taking the nUlnber of copies of elernents into account. The $\cup$ operation simply adds up the nurnber of copies of an elernent, the $\cap$ operation counts the lesser nU1nbel' of tirnes a given elernent appears in the two input rnultisets, and $-$ subtracts the nurnber of ti1nes a given e1ernent appears in the second lnultiset frorn the nUlnber of tinles it appears in the first Inultiset. For exarnplc, using rnultiset scrnantics $\cup(\{1,2,2,2\}, \{2,2,3\}) = \{1,2,2,2,2,2,3\}$, $\cap(\{1,2,2,2\}, \{2,2,3\}) = \{2,2\}$; and $-(\{1,2,2,2\}, \{2,2,3\}) = \{1,2\}$.

Lists

Traditional list operations include *head*, \vhich returns the first ele1nent; *tail*, which returns the list obtained by rCIlloving the first elernent; *prepend*, which

2Note that the first element in an SQL array has index value 1 (not 0, as in sorne languages).

takes an elcrnent and inserts it as the first elernent in a list; and *append*, which appends one list to another.

23.3.4 Queries Over Nested Collections

We no\v present SOHle exar11ples to illustrate ho\v relations that contain nested collections can be queried, using SQL syntax. In particular, extensions of the relational rnodel with nested sets and rnultisets have been \videly studied and \ve focus on these collection types.

We consider a variant of the FihIIS relation from Figure 23.1 in this section, with the *stars* field defined as a setof (VARCHAR[25]), rather than an array. Each tuple describes a filrn, uniquely identified by *filrnno,* and contains a set (of stars in the filrn) as a field value.

Our first exarnple illustrates how we can apply an aggregate operator to such a nested set. It identifies filrns with r11or8 than two stars by counting the nurnber of stars; the CARDINALITY operator is applied once per FilnIs tuple. [3]

```
SELECT  F.filmno
FROM    Filrns F
WHERE   CARDINALITY(F.stars) > 2
```

Our second query illustrates an operation called unnesting. Consider the instance of Filrns shown in Figure 23.4; we have olnitted the *director* and *budget* fields (included in the Filnls schema in Figure 23.1) for simplicity. A flat version of the saIne inforrna.tion is shown in Figure 23.5; for each filrn and star in the £ibn, we have a tuple in Filrns_flat.

filmno	*title*	*stars*
98	Casablanca	{Bogart, Bergluan}
54	Earth vVorrns Are Juicy	{Herbert, Wanda}

Figure **23.4** A Nested Relation, Films

1"11e follc)\ving query generates the instance of Films_flat from Fihns:

```
SELECT  FJilrnno, F.title, S AS star
FROM    FilrnsF, F.stars AS S
```

[3] SQL:1999 does not support set or rnultiset values, as we noted earlier. If it did, it would be natural to allow the CARDINALITY operator to be applied to a set-value to count the nUluber of elements; we have used the operator in this spirit.

filmno	*title*	*star*
98	Casablanca	Bogart
98	Casablanca	Bergman
54	Earth Worms Are Juicy	Herbert
54	Earth Worms Are Juicy	Wanda

"Figure 23.5 A Flat Version, Films_flat

The variable *F* is successively bound to tUI>les in Filrns, and for each value of *F*, the vaTiable *S* is successively bound to the set in the *stars* field of *F*. Conversely, we Inay want to generate the instance of F'ilrns frorn FilIns_fiat. We can generate the Filrns instance using a, generalized fonn of SQL's GROUP BY C011struct, as the following query illustrates:

```
SELECT    F.filmno, F.title, set_gen(F.star)
FROM      Fihns.Jlat F
GROUP BY  F.fihnno, F.title
```

This oxaluplc introduces a new operator *set_gen*, to be used with GROUP **BY**, that requires sorne explanation. The GROUP BY clause partitions the Films_flat table by sorting on the *filmno* attribute; all tuples in a given partition have the sanle *filrnno* (and therefore the sarne *title*). Consider the set of values in the *star* cohunn of a given partition. In an SQL-92 query, this set rnust be surnluarized by applying an aggregate operator such as COUNT. Now that we allow relations to contain sets as field values, however, we can return the set of *star* values as a field value in a single answer tuple; the answer tuple also contains the *filmno* of the corresponding partition. rrhe *set_gen* operator collects the set of *star* values in a paTtition and creates a set-valued object. This operation is called nesting. We can irnagine similar generator functions for creating Inuitise1's, lists, and so on. However, such generators are not included in SQL:1999.

23.4 ENCAPSULATION AND ADTS

Consicler the Frames table of Figure 23.1. It has a colun111 *image* of type `jpeg_image`, which stores a cOlnpressed image representing a single frarne of a film. The jpeg_image tYI)C is not one of the DBMS's l>uilt-in types and was defined 1)y a user for the I)inky application to store ilna,ge data cornpressed using the JPEG stanclard. As another exarllple, the Countries table defined in Line 7 of Figure 23.1 has a colurnn *boundary* of t,ype polygon, which contains representations of the shapes of countries' outlines on a world rnap.

.Allowing users to define arbitrary new data types is a key feature of ORDBMSs. The DBMS allows users to store and retrieve objects of type `jpeg_image`, just like an object of any other type, such as **integer**. New atomic data types usually need to have type-specific operations defined by the user who creates thern. For example, one rnight define operations on an irnage data type such as compress, **rotate**, **shrink**, and crop. The ccnnbination of an atolIlic data type and its associated rnethods is called an **abstract data type**, or ,A,DT. Traditional SQL COlnes with built-in ADTs, such as integers (with the associated arithnletic rnethods) or strings (with the equality, cornparison, and LIKE llIethods). Object-relational systerns include these ADT's and also allow users to define their o\vn ADTs.

The label *abstract* is applied to these data types because the database systerIl does not need to know how an ADT's data is stored nor ho\v the ADT's rnethods work. It rnerely needs to know what rnethods are availa,ble and the input and output types for the rnethods. I-Eding ADT internals is called **encapsulation**.[4] Note that even in a. relational systern, atolnic types such as integers have associated rnethods that encapsulate the1n. In the case of integers, the standard Inethods for the ADT are the usual aritlunetic operators and COlll-parators. To evaluate the addition operator on integers, the database systenl need not understand the laws of addition it l11erely needs to know how to invoke the addition operator's code and what type of data to expect in return.

In an object-relational systenl, the Silllplification due to encapsulation is critical because it hides any substantive distinctions between data types and allows an OR,DB1VIS to be iInplernented \vithout anticipating the types and rnethods that users Inight want to add. For exarnple, (l,dding integers and overlaying irnages can be treated unifonnly by the systern, with the only significant distinctions being that different code is invoked for the two operations and differently typed objects are expected to be returned froIll that code.

23.4.1 Defining Methods

To register a new rnethod for a user-defined data type, users rnust write the code for the nlcthod and then infor1n the database systcrI1 about the Inethod. The cod.e to be written depends on the languages supported by the DBlVIS and, possibly, the operating systerH in question. For example, the ORDBMS Inay handle Java code in the Linux operating systern. In this case, the lnet,hod code nlu,st be written iII Java and cOlnpiled into a Java bytecode file stored in. a Linux file system. 'Then an SQL-style luethod registration eOllunand is given to the ()I:ll)BlVlS so that it recognizes the new rnethod:

[4]Some ORDBMSs actually refer to ADTs as **opaque types** because they are encapsulated and hence one cannot see their details.

Packaged ORDBMS Extensions: Developing a set of user-defined types and rnethods for a particular application-say image management—can involve a significant aIIlount of work and dornain-speeific expertise. As a result, most ORDBMS vendors partner with third parties to sell prepackaged sets of ADrrs for particular domains. Inforn1ix calls these extensions *DataBlades,* Oracle calls theln *Data Cartridges,* IBM calls thern *DB2 Extenders,* and so on. These packages include the ADT 11lethod code, DDL scripts to automate loading the ADTs into the system, and in some cases specialized access methods for the data type. Packaged ADT extensions are analogous to the class libraries available for object-oriented programIning languages: They provide a set of objects that together address a COlnnlon task.

SQL:1999 has an extension called SQL/MIVI that consists of several independent parts, each of which specifies a type library for a particular kind of data. The SQL/MM parts for Full-Text, Spatial, Still Iillage, and Data Mining are available, or nearing publication.

```
CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
```

This statenlent defines the salient aspects of the lllethod: the type of the associated ADT, the return type, and the location of the code. Once the method is registered, the DBMS uses a Java, virtual lnachine to execute the code[5]. Figure 23.6 presents a nUlnber of rnethod registration cOllllnands for our Dinky database.

```
1. CREATE FUNCTION thumbnail(jpeg_image) RETURNS jpeg_image
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
2. CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
        AS EXTERNAL NAME '/a/b/e/dinky.class' LANGUAGE 'java';
3. CREATE FUNCTION isJnerbert(jpeg_image) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
4. CREATE FUNCTION radius (polygon, float) RETURNS polygon
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
5. CREATE FUNCTION overlaps (polygon, polygon) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
```

Figure 23.6 Method Registration Conunands for the Dinky Database

5In the case of non-portable cOIl1piled code written, for example, in a language like C++"—the D131v18 uses the operating system's dynamic linking facility to link the method code into the database system so that it can be invoked.

rrype definition statelnents for the user-defined atornic data types in the Dinky scherna are given in Figure 23.7.

```
1. CREATE ABSTRACT DATA TYPE jpeg_image
     (internallength = VARIABLE, input = jpeg_in, output = jpeg_out);
2. CREATE ABSTRACT DATA TYPE polygon
     (internallength = VARIABLE, input = polyjn, output = poly_out);
```

Figure 23.7 Atomic Type Declaration Commands for Dinky Database

23.5 INHERITANCE

We considered the concept of inheritance in the context of the ER, model in Chapter 2 and discussed how ER diagrarns with inheritance 'were translated into tables. In object-database systems, unlike relational systerns, inheritance is supported directly and allows type definitions to be reused and refined very easily. It can be very helpful when modeling similar but slightly different classes of objects. In object-database systerns, inheritance can be used in two ways: for reusing and refining types and for creating hierarchies of collections of sirnilar but not identical objects.

23.5.1 Defining Types with Inheritance

In the Dinky database, we rnodel rnovie theaters with the type theater_t. Dinky also wants their database to represent a new rnarketing technique in the theater business: the *theater-cafe,* which serves pizza and other rneals while screening movies. rrheater-cafes require additional inforrnation to be represented in the database. In particular, a theater-cafe is just like a theater, but has an additional attribute representing the theater's IIlenu. Inheritance allows us to capture this 'specialization' explicitly in the database design with the followiIlg DDL staternent:

```
CREATE TYPE theatercafe_t UNDER theater_t (rn,enu text);
```

This staternent creates a new type, theatercafe_t, which has the sarne attributes and rnethods as theater_t, plus one additional attribute *menu* of type text. Methods defined on theater_t apply to objeets of type theatercafe_t, but not viee versa. We say that theatercafe_t inherits the attributes and rnethods of theater_t.

Note that the illherita,nce rnechanisrll is not rnerely a rnacro to shorten CREATE staternents. It creates an explicit relationship in the database between the subtype (theatercafe_t) and the supertype (theater_t): *An object of the* *subtype is also considered to be an object of the supertype.* This treatment Ineans that an.y operations that apply to the supertype (nlcthods as well as query operators, such as projection or join) also apply to the subtype. This is generall.y· expressed in the follo\ving principle:

> **The Substitution Principle:** Given a supertype A and a subtype B, it is always possible to substitute an object of type B into a legal expression written for objects of type A, without producing type errors.

This principle enables easy code reuse because queries and Inethodswritten for the supertype can be applied to the subtype without I1lodification.

Note that inheritance can also be used for atomic types, in addition to ro\v types. Given a supertype image_t with rnethods *title()*, *number_of_colors()*, and *d'isplay()*, we can define a subtype thumbnail_image_t for slllall irnages that inherits the rnethods of image_to

23.5.2 Binding Methods

In defining a subtype, it is sornetiInes useful to replace a rnethod for the sII-pertype with a new version that operates differently on the subtype. Consider the image_t type and the subtype jpeg_image_t frorH the Dinky database. lJnfortunately, the *display()* rnethod for standard images does not work for JPEG irnages, which are specially cOlnpressed. Therefore, in creating type jpeg_image_t, we write a special *display()* rnethod for JPEG iruages and register it with the database systern using the CREATE FUNCTION cOIIuuand:

```
CREATE FUNCTION display(jpeg_image) RETURNS jpeg_image
        AS EXTERNAL NAME '/a/b/c/jpeg.class' LANGUAGE 'java.';
```

Registering a new rnethod with the sarne narne as an old rnethod is called **overloading** the luethod narne.

Because of overloading, the systern Inust understand 'which rnethod is intended in a particular expression. For example, when the systern needs to invoke the *display*() rnethod on an object of type j peg_image_t, it uses the specialized *display* rnethocL When it needs to invoke *display* on an object of type image_t that is not otherwise subtyped, it invokes the standard *display* Inethod. The process of deciding which rnethod to invoke is called **binding** the rnethod to the object. In certain situations, this binding can be done when an expression is parsed (**early binding**), but in other cases the 1n08t specific type of an object cannot be known until rl.ln-tinle, so the rnethod cannot be l)ound until then (**late binding**). Late birlding fa,cilties acId flexibility but can rnake it harder for the user to reason about the Inethods that get invoked for a given query expreSSIon.

23.5.3 Collection Hierarchies

Type inheritance was invented for object-oriented progranuning languages, and our discussion of inheritance up to this point differs little £roln the discussion one Inight find in a book on an object-oriented language such as C++ or Java.

However, because database systerns provide query languages over tabular data sets, the lnechanisnls fronl programming languages are enhanced in object databases to deal with tables and queries as well. In particular, in objeet-relational systelIls, we can define a table containing objects of a particular type, such as the Theaters table in the Dinky sehenla. Given a new subtype, such as theatercafe_t, we would like to create another table Theater_cafes to store the inforrnation about theater cafes. But, when writing a query over the Theaters table, it is sornetirnes desirable to ask the saIne query over the rrheater_cafes table; after all, if we project out the additional C01UIIII1S, an instance of the Theater_cafes table can be regarded as an instance of the Theaters table.

R,ather than requiring the user to specify a separate query for each such table, we can infonn the systern that a new table of the subtype is to be treated as part of a table of the supertype, with respect to queries over the latter table. In our exalnple, we can say

```
CREATE TABLE Thea,ter_Cafes OF TYPE theatercafe_t UNDER Theaters;
```

This staternent tells the systern that queries over the Theaters table should actually be run over all tuples in both the rrheaters and rrheater_Cafes tables. In such cases, if the subtype definition involves rnethod overloading, late-binding is used to ensure that the appropriate rnethods are called for each tuple.

In general, t11.e UNDER clause can be used to genera,te an arbitrary tree of tables, called (1, **collection hierarchy**. Queries over a particular tal)le T in the hierarchy are run over all tuples in rr and its descendants. Sornetirnes, a user rnaywant the query to nUI only on rr and not on the descendants; additiona1 syntax, for exaInple, the key\vord ONLY, can be used in the query's FROM clause to achieve this effect.

23.6 OBJECTS, OIDS, AND REFERENCE TYPES

In object-elatabase systerns, data objects can be given an **object identifier (oid)**, \which is sotne value that is unique in the database across tirne. The

OIDs: IBM DB2, Informix lJDS, and Oracle 9i support REF types.

DBMS is responsible for generating aids and ensuring that an oid identifies an object uniquely over its entire lifetime. In SOHle systenls, all tuples stored in any table are objects and autornatically assigned unique oids; in other systenls, a user can specify the tables for 'which the tuples are to be assigned aids. Often, there are also facilities for generating oids for larger structures (e.g., tables) as well as slnaller structures (e.g., instances of data values such as a copy of the integer 5 or a .TPEG image).

An object's aid can be used to refer to it from elsewhere in the data. An oid has a type similar to the type of a pointer in a progralnll1ing language.

In SQL:1999 every tuple in a table can be given an aid by defining the table in ternlS of a structured type and declaring that a REF type is associated with it, as in the definition of the Theaters table in Line 4 of Figure 23.1. Contrast this with the definition of the Countries table in Line 7; Countries tuples do not have associated aids. (SQL:1999 also assigns 'oids' to large objects: This is the locator for the object.)

REF types have values that are unique identifiers or aids. SQL:1999 requires that a given REF type must be associated with a specific table. For exalnple, Line 5 of Figure 23.1 defines a cohllnn *theater* of type REF(theater_t). The SCOPE clause specifies that iterns in this colurnn are references to rows in the rrheaters table, which is defined in Line 4.

23.6.1 Notions of Equality

The distinction between reference types and reference-free structured types raises another issue: the definition of equality. Two objects having the saIne type are defined to be **deep equal** if and only if

1. The objects <1,1'e of atolnic type and have the same value.
2. The objects are of reference type and the *deep equals* operator is true for the two referenced objects.
3. The objects are of structured type and the *deep equals* operator is true for all the corresponding subparts of the two objects.

Two objects that have the same reference type are defined to be **shallow equal** if both refer to the saIne object (i.e., both references use the saUle aid). The definition of shallow equality can be extended to objects of arbitrary type by taking the definition of deep equality and replacing *deep equals* by *shallow equals* in parts (2) and (3).

As an example, consider the cornplex objects ROW(538, *tS9,* 6-3-97, 8-7-97) and ROW(538, *i33,* 6-3-97, 8-7-97), whose type is the type of rows in the table Nowshowing (Line 5 of Figure 23.1). 1'hese two objects are not shallow equal because they differ in the second attribute value. Nonetheless, they rnight be deep equal, if, for instance, the oids *t89* and *t33* refer to objects of type theater_t that have the saIne value; for exarnple, tuple(54, 'Majestic', '115 King', '2556698').

While two deep equal objects Inay not be shallow equal, as the exarnple illustrates, two shallow equal objects are always deep equal, of course. 'The default choice of deep versus shallow equality for reference types is different across systenls, although typically we are given syntax to specify either semantics.

23.6.2 Dereferencing Reference Types

An item of reference type REF(basetype) is not the sarne as the basetype itenl to which it points. To access the referenced basetype itenl, a built-in deref() rnethod is provided along with the REF type constructor. For example, given a tuple from the Nowshowing table, one can access the *name* field of the referenced theater_t object with the syntax Nowshowing.deref*(theater). narne.* Since references to tuple types are comInon, SQL:1999 uses a Java-style arrow operator, which cOD.lbines a postfix version of the dereference operator with a tuple-type dot operator. The narne of the referenced theater can be accessed with the equivalent syntax *Nowshowing.theater-> narne,* as in Figure 23.3.

At this point we have covered all the basic type extensions used in the Dinky scherna in Figure 23.1. The reader is invited to revisit the scherna and exarnine the structure and content of each table and how the new features are used in the various sarnple queries.

23.6.3 URLs and DIDs in SQL:1999

It is instructive to note the differences between Internet lJRIJs and the oids in object systerns. First, oids uniquely identify a single object over all tirne (at least, until the object is deleted, when the oid is undefined), whereas the Web resource pointed at by an URL can change over tirue. Second, oids are sirnply identifiers and carry no physical infonnation about the objects they identify this rnakes it possible to change the storage location of an object without rnodifying pointers to the object. In contrast, URLs include network addresses and often file-syst;enl names as well, meaning that if the resource identified by the URL has to move to another file or network address, then all links to that resource are either incorrect or require a 'forwarding' rnechanisH1. Third, oids are automatically generated by the DBMS for each object, whereas URLs are user-generated. Since users generate URLs, they often ernbed sc-rnantic inforlllation into the URL via rnachine, directory, or file names; this can becoine confusing if the object's properties change over tilne.

For URLs, deletions can be troublesorne: This leads to the notorious '404 Page Not Found' error. For oids, SQL:1999 allows us to say REFERENCES ARE CHECKED as part of the SCOPE clause and choose one of several actiol1swhen a referenced object is deleted. This is a direct extension of referential integrity that covers oids.

23.7 DATABASE DESIGN FOR AN ORDBMS

The rich variety of data types in an ORDBMS offers a database designer Inany opportunities for a rnore natural or lllore efficient design. In this section we illustrate the differences between RDBMS and ()RI)BMS database design through several exarnples.

23.7.1 Collection Types and ADTs

()ur first exarnple involves several space probes, each of which continuously records a video. A single video strearll is associated with each probe, and while this strearn was conected over a certail1 tiule period, we assurne that it is now a cOlllplete object associated with the probe. During the tirne period over which the video was collected, the probe's locatiol1\vas periodieaJly recorded (such infonnation can easily be piggy-backed onto the header portion of a video streanl conforrning to the MPEG sta,ndard). The inforrnation associated with a probe has three parts: (1) a *probe ID* that identifies a probe uniquely, (2) a *video stream*, and (3) a *location sequence* of ⟨*time, location*⟩ pairs. What kind of a database scherna should we use to store this infonnation?

An RDBMS Database Design

In an RDBMS, we rnust store each video strcanl as a BLO13 and each location sequellce as tuples in a table. A possible RDBMS database design follo\vs:

Probes(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`,
camera: `string`, *video:* BLOB)

There is a single table caned Probes and it has several rows for each probe. Each of these rows has the same *pid*, *camera* and *video* values, but different *time*, *tat*, and *long* values. (We have used latitude and longitude to denote location.) The key for this table can be represented as a functional dependency: $IJTLN \rightarrow CV$, where N stands for longitude. There is another dependency: $P \rightarrow CV$. This relation is therefore not in BCNF; indeed, it is not even in 3NF. We ca:n decolupose Probes to obtain a BCNF scherna:

Probes_Loc(*pid:* integer, *time:* timestamp, *tat:* real, *long:* real)
Probes_Video(*pid:* integer, *camera:* `string`, *video:* BLOB)

This design is about the best we can achieve in an RDBMS. However, it suffers frorn several drawbacks.

First, representing videos as BLOBs lI1eanS that we have to write application code in an external language to lnanipulate a video object in the database. Consider this query: "For probe 10, display the video recorded between 1:10 P.M. and 1:15 P.M. on May 10 1996." We lnust retrieve the entire video object associated "lith probe 10, recorded over several hours, to display a segrnent recorded over five rninutes.

Next, the fact that each probe has an associated sequence of location readings is obscured, and the sequence infornmatiol1 associated with a probe is dispersed across several tuples. A third drawback is that we are forced to separate the video infonnation froTn the sequence infornation for a probe. These lirnitations are exposed by queries that require us to consider all the infonnation associated with each probe; for example, "For each probe, print the earliest tirne at which it recorded, and the camera type." T'his query now involves a join of Probes_Loc and Probes_Video on the *pid* field.

An ORDBMS Database Design

;\n ORDBMS supports a lnuch better solution. First. we can store the video as an A.DT object and write rnethods that capture a.ny special rna,nipulation we wish to perforrr1. Second. because we are allowed to store structured types such as lists, we (:a,n stc)re the location sequence for a probe in a single tuple, along\vith the video infonnation. This layout eliminates the need for joins in queries that involve both the sequence and video infornation. An ORDBMS design for our example consists of a single relation called Probes_AllInfo:

Probes_AllInfo(*pid:* integer, *locseq:* location_seq, *camera:* string;
video: `mpeg_stream`)

This definition involves two new types, **location_seq** and mpeg_stream. The mpeg_stream type is defined as an ADT, with a Inethod *display()* that takes a start time and an end tirne and displays the portion of the video recorded during that interval. This rnethod can be irnplernented efficiently by looking at the total recording duration and the total length of the video and interpolating to extract the segnlcnt recorded during the interval specified in the query.

Our first query in extended SQL using this *display* lnethod follows. We now retrieve only the required segment of the video rather than the entire video.

```
SELECT  display(P.video, 1:10 PoM May 10 1996, 1:15 PoM May 10 1996)
FROM    Probes_AllInfo P
WHERE   Popid = 10
```

Now consider the location_seq type. We could define it as a list type, containing a list of ROW type objects:

```
CREATE TYPE location_seq listof
        (row (time: timestamp, lat: real, long: real))
```

Consider the *locseq* field in a row for a given probe. This field contains a list of rows, each of which has three fields. If the ORDBMS implements collection types in their full generality, we should be able to extract the *time* colurnn from this list to obtain a list of timestamp values and apply the MIN aggregate operator to this list to find the earliest time at which the given probe recorded. Such support for collection types would enable us to express our second query thus:

```
SELECT   P.piel, MIN(P.locseq.tirne)
FROM     Probes._AllInfo P
```

Current ORDBMSs are not as general and clean as this exalnple query suggests. For instance, the systern rnay not recognize that projecting the *time* colurnn frorn a list of rows gives us a list of tirnestarnp values; or the systcru rnay allow us to apply an aggregate operator only to a table and not to a nested list value.

Continuing with our example, we lnay want to do specialized operations on our location sequences that go beyond the standard aggregate operators. For instance, we rnay want to define a lnethod that takes a tirne interval and COIIl-putes the distance traveled by the probe during this interval. The code for this rnethod rnust understand details of a probe's trajectory and geospatial coordinate systenls. FbI' these reasons, we rnight choose to define location_seq as an ADT\.

Clearly, an (ideal) ORDBMS gives us IllaIlY useful design options that are not available in an RDBMS.

23.7.2 Object Identity

We now discuss S0l11e of the consequences of using reference types or aids. The use of aids is especially significant when the size of the object is large, either because it is a structured data type or because it is a big object such as an image.

Although reference types and structured types seem sirnilar, they are actually quite different. For example, consider a structured type my_theater tuple *(ina* integer, *name* text, *address* text, *phone* text) and the reference type theater ref (theater_t) of Figure 23.1. rrhere are irnportant differences in the way that database updates affect these two types:

- Deletion: Objects with references can be affected by the deletion of objects that they reference, while reference-free structured objects are not affected by deletion of other objects. For exaluple, if the Theaters table were dropped from the database, an object of type theater might change value to *null*, because the theater_t object it refers to has been deleted, while a similar object of type my_theater would not change value.
- Update: Objects of reference types change value if the referenced object is updated. Objects of reference-free structured types change value only if updated directly.
- Sharing versus Copying: An identified object can be referenced by llluitiple reference-type iterIls, so that each update to the object is reflected in IYlany places. To get a sirnilar effect in reference-free types requires updating all 'copies' of an object.

There are also irnportant storage distinctions between reference types and non-reference types, which rnight affect perfoI'rnance:

- Storage Overhead: Storing copies of a large value in rnultiple structured type objects IYlay use lnnch mo1'e space than stori.ng the value once and referring to it elsewhere through reference type objects. This additional storage requirelnent can affect both disk usage and buffer lnanagcrncnt (if Illal1Y copies are accessed at once).
- Clustering: The subparts of a structured object are typically stored together on disk. Objects with references ma,Y point to other objects that are far a:way on the disk, and the disk arm Inay require significant mOVClnent

OIDs and Referential Integrity: In SQL:1999, all the oids that appear in a cohunn of a relation are required to reference the same target relation. This 'scoping' makes it possil)]e to check oid refere:nces for 'referential integrity' just like foreign key references are checked. While current ORDBMS products supporting oids do not support such checks, it is likely that they will in future releases. This will nlake it rnnch safer to use aids.

to asserrlble the object and its references together. Structured objects can thus be l'nor8 efficient than reference types if they are typically accessed in their entirety.

Many of these issues also arise in traditional prograunuing languages such as C or Pascal, which distinguish between the notions of referring to objects *by value* and *by reference.* In database design, the choice between using a structured type or a reference type typically includes consideration of the storage costs, clustering issues, and the effect of updates.

Object Identity versus Foreign Keys

lJsing an oid to refer to an object is silnila,r to using a foreign key to refer to a tuple in another relation but not quite the same: An oid can point to an object of theater_t that is stored *anywhere* in the database, even in a field, whereas a foreign key reference is constrained to point to an object in a particular referenced relation. This restriction rnakes it possible for the DBMS to provide lnuch greater support for referential integrity than for arbitra,ry aid pointers. In general, if an object is deleted while there are still oid-pointers to it, the best the DBl\IIS can do is to recognize the situation by rnaintajning a reference count. (Even this lirnited support becornes irnpossible if oids can be copied freely.) Therefore, the responsibility for avoiding dangling references rests largely with the user if oids are llsed to refer to objects. This burdensoIllc responsibility suggests that we should use oids with great ca.ution and use foreign keys instead \vhenever possible.

23.7.3 Extending the ER Model

The ER rnodel, as described in Chapter 2, is not adequate for ORDBMS design. We have to use an extended ER rnodel that supports structured attributes (i.e., sets, lists, arra,Ys as attribute values), distinguishes whether entities have ol)ject ids, and allows us to Inodel entities whose attributes include rnethods. We illustrate these connnents using an extended ER diagrarn to describe the space probe data in Figure 23.8; our notational conventions are ad hoc and only for illustrative purposes.

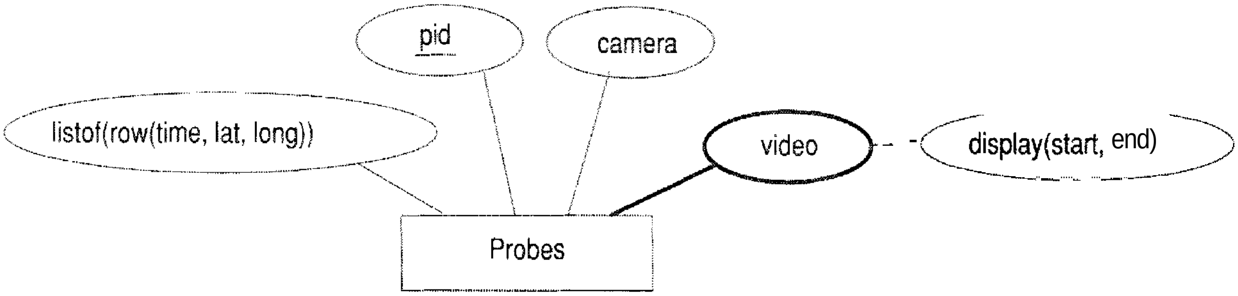


Figure 23.8 The Space Probe Entity Set

The definition of Probes in Figure 23.8 has two new aspects. First, it has a structured-type attrilnlte **listof** (row(*time, lat, lo'ng)); each* value assigned to this attribute in a Probes entity is a list of tuples with three fields. Second, Probes has an attribute called *video* that is an abstract data type object, \vhich is iIldicatecl by a dark oval for this attribute with a dark line connecting it to Probes. Further, this attribute has an 'attribute' of its own, which is a rnethod of the *J\DT.*

Alternatively, we could rnodel each video as an entity by using an entity set called Videos. The association between Probes entities and Videos entities could then be captured by defining a relationship set that links thenl. Since each video is collected by precisely one probe and every video is collected by se)lne probe, this relationship can be rna,intained by simply storing a reference to a probe object with each Videos entity; this technique is essentially the second translation approach frorn ER, diagrams to tables discussed in Section 2.4.1.

If we also rnake Videos a weak entity set in this alternative design, we can add a referential integrity constraint that causes a Videos entity to be deleted \vhen the corresponding Probes entity is deleted. More generally, this alternative design illustrates a strong sirnila1'ity between storing references to objects and foreign keys; the foreign key mechanisT11 achieves the saIne effect as storing oids, but in a controlled lUanneI'. If oids are used. the user rnusi ensure that there are no dangling references when an object is deleted, with very little support froIll the DBMS.

Finally, we note tllat a significant extension to the ER rHodel is required to support the design of nested collections. For example, if a location sequence is rnodeled as an entity, and we want to define an attribute of Probes that contains a set of such entities, there is no way to do this\vithont extending the ER model. We do not disC1ISS this I)oint furtller at the level of ER diagrams, but consider an exaJnple next that illustrates when to use a nest,ed collection.

23.7.4 Using Nested Collections

Nested collections offer great rnodeling power but also raise difficult design decisions. Consider the following way to rnodel location sequences (other infonnation about probes is ornitted here to sirnplify the discussion):

Probesl(pid: integer, *locseq:* location_seq)

rrhis is a good choice if the irnportant queries in the workload require us to look at the location sequence for a particular probe, as in the query "For each probe, print the earliest tirne at which it recorded and the caluera type." On the other hand, consider a query that requires us to look at all location sequences: "Find the earliest time at which a recording exists for *laf;=5, long=90."* This query can be answered 1110re efficiently if the following scherna is used:

Probes2(*pid:* integer, *time:* timestamp, *tat:* real, *long:* real)

The choice of scherna Blust therefore be guided by the expected workload (as always). As another example, consider the following scherna:

Can_Teachl *(cid:* integer, *teacheTs:* setof *(ssn:* string), *sal:* integer)

If tuples in this table are to be interpreted as "Course *cid* can be taught by any of the teachers in the *teacheTs* field, at a cost *sal."* then we have the option of using the following schenla. instead:

CarLTeach2(*cid:* integer, *teachCT_8sn:* string, *sal:* integer)

A choice between these two alternatives can be Inade based on how we expect to query this table. On the other hand, suppose that tuples in CalL.Teachl are to be interpreted as "Course *cid* can be taught by the *tearnteacheT8,* at a cornbined cost of *sal."* CarLTeach2 is no longer a viable alternative. If we wanted to flatten Can_Teach1, we would have to use (1, separate table to encode tearns:

Can_Teach2(*cid:* integer, *team_id:* oid, *sal:* integer)
Teams(*tid:* oid, *ssn:* string)

As these exarnples illustrate, nested collections are appropriate in certain situations, but this fea,ture can easily be rnisused; nested collections should therefore be used ·with care.

23.8 ORDBMS IMPLEMENTATION CHALLENGES

The enhanced functionality of ORDBMSs raises several irnp1ernentation challenges. SOlne of these are 'well understood and solutions have been irnp1enlented in products; others are subjects of current research. In this section \ve exarnine a few of the key challenges that arise in irnplernenting an efficient, fully functional ORDBMS. Many rnore issues are involved than those discussed here; the interested reader is encouraged to revisit the previous chapters in this book and consider whether the irnplernentation techniques described there apply naturally to ORDBMSs or not.

23.8.1 Storage and Access Methods

Since object-relational databases store new types of data, ORDBMS implernentors need to revisit some of the storage and indexing issues discussed in earlier chapters. In particular, the system lllust efficiently store ADT objects and structured objects and provide efficient indexed access to both.

Storing Large ADT and Structured Type Objects

Large ADT objects and structured objects cornplicate the layout of data on disk. This problern is well understood and has been solved in essentially all ORDBMSs and OODBMSs. We present Sallie of the main issues here.

User-defined ADTs can be quite la,rge. In particular, they can be bigger than a single disk page. Large ADTs, like BLOBs, require special storage, typically in a different location on disk frorn the tuples that contain them. Disk-based pointers are rnaintained frorn the tuples to the objects they contain.

Structured objects can also be large, but unlike ADrr objects, they often vary in size during the lifetirne of a database. For exarnple, consider the *stars* attribute of the *film,s* table in Figure 23.1. As the years pass, SOlne of the 'bit actors' in an old rnovie may becorne famous.[6] When a bit actor hecornes farnous, Dinky rnight want to advertise his or her presence in the earlier filrns. This involves an insertion into the *stars* attribute of an individual tuple in *filrns*. Because these bulk attributes can grow arbitrarily, flexible disk layout rnechanisrns are required.

[6] A famous example is Marilyn Monroe, who had a bit part in the Bette Davis classic *All About Eve*.

An additional COlllplication arises \'lith array types. Traditionally, array elelnents are stored sequentially on disk in a row-by-row fashion; for example

$$A_{11},\ldots,A_{1n},A_{21},\ldots,A_{2n},\ldots,A_{m1},\ldots,A_{mn}$$

However, queries rnay often request suba.rrays that are not stored contiguously on disk (e.g., $A_{11}, A_{21}, \ldots, A_{rn1}$). Such requests can result in a very high I/O cost for retrieving the subarray. To reduce the nurnber of l/Os required, arrays are often broken into contiguous *chunks*, which are then stored in some order on disk. Although each chunk is sorne contiguous region of the array, chunks need not be row-by-row or colurnn-by-colurllll. For exalnple, a chunk of size 4 111ight be All, A_{12}, A_{21}, A_{22}, 'which is a square region if we think of the array as being arranged row-by-row in two dimensions.

Indexing New Types

One ilnportant reason for users to place their data in a database is to allow for efficient access via indexes. Unfortunately, the standard RDBMS index structures support only equality conditions (B+ trees and hash indexes) and range conditions (B+ trees). An irnportant issue for OR,DB1\IISs is to provide efficient indexes for AD'I' rnethods and operators on structured objects.

Many specialized index structures have been proposed by researchers for particular applications such as cartography, genorne research, 11ultirnedia repositories, Web search, and so on. An ORDBMS cornpany cannot possibly inlplernent every index that has been invented. Instead, the set of index structures in an ORDBMS should be user-extensible. Extensibility would allow an expert in cartography, for exanlple, to not only register an ADT for points on a rnap (i.e., latitude··longitude pairs) but also irnplernent an index structure that supports natural rnap queries (e.g., the R-tree, \vhich lnatches cOllclitions such as “Find rne all theaters within 100 Iniles of Andorra"). (See Chapter 28 for 1110re on R-trees and other spatial indexes.)

One way to rnake the set. of index structures extensible is to publish an *access method interface* that lets users irnplcrnent an index structure *outside* the DBMS. The index and data can be stored in a file systeIll and the DBMS sirnply issues the *open*, *next*, and *close* iterator requests to the user's external index code. Such functionality rnakes it possible for a user to connect a I)Bl\1S to a Web search engine, for example. A rnain drawback of this approach is that data in an external index is 11ot protected by the])B1V1S'8 support for concurrency and recovery. 1\n alternative is for the ORDBMS to provide a generic 'template' index structure that is sufficiently general to encornpass rnost index structures that usersrn.ight invent. Because snell a structure is implemented within the DBMS, it can support high concurrency and recovery. The *Gener-*

alized Search Tree (GiS'r) is suell a structure. It is a ternplate index structure based on B+ trees, \vhich aJlo\vs IIIOSt of the tree index structures invented so far to be irnplernented'with only a few lines of user-definedAD'T' code.

23.8.2 Query Processing

ADTs and structured types call for l1e\v functionality in processing queries in ORDBMSs. They also change a nurnber of assumptions that affect the efficiency of queries. In this section we look at two functionality issues (u8er-defined aggregates and security) and two efficiency issues (rnethod caching and pointer swizzling).

User-Defined Aggregation Functions

Since users are allowed to define new rnethods for their ADTs, it is not unreasonable to expect thern to want to define new aggregation fUllctions for their ADTs as well. For example, the usual SQL aggregates····---CDUNT, SUM, MIN, MAX, AVG--are not particularly appropriate for the image type in the Dinky schema.

Most ORDBMSs allow users to register new aggregation functions \vith the systern. To register an aggregation function, a user lnust iruplenlent three methods, which we call *initialize*, *iterate,* and *terrninate*. The *initialize* rnethod initializes the internal state for the aggregation. The *iterate* rnethod updates that state for every tuple seen, while the *terrninate* rnethod C0111putes the aggregation result based on the final state and then cleans up. As an exarnple, consider an aggregation function to cornpute the second-highest value in a field. The *initialize* call would allocate storage for the top two values, the *iterate* call would corupare the current tuple's value with the top two and update the top two as necessary, and the *terminate* call \vould delete the storage for the top two values, returning a copy of the second-highest value.

Method Security

ADTs give users the power to add code to the DBMS; this power can be abused. A buggy or rnalicious ADT rnethod can bring do\vn the database server or even corrupt the database. The DBMS lnust have rnechanisrns to prevent buggy or rnalicious user code from causing probleIlls. It 1nay rnake sense to overricle these rnechanislIls for efficiency in production environrnents with vendor-supplied rnethods. I-Io\vever, it is irnportant for the rnechanisrns to exist, if only to support delJugging of *J\DT* rnethocls; otherwise rnethod writers

\vould have to \vrite bug-free code before registering their rnethods with the DBMS--not a very forgiving prograIIuning environlnent.

()ne rnechanisrn to prevent problerns is to have the user rnethods be *interpreted* rather than *compiled*. The DBMS can check that the rnethod is well behaved either by restricting the power of the interpreted language or by ensuring that each step taken by a rnethod is safe before executing it. Typical interpreted la.n-guages for this purpose include Java and the procedural portions of SQL:1999.

An alternative rnechanislll is to allow user methods to be cOlnpiled frorn a general-purpose progranuning language, such as C++, but to run those rneth-ods in a different address space than the DBMS. In this case, the DBMS sends explicit interprocess cOl1uIlunications (IPCs) to the user method, which sends IPCs back in return. This approach prevents bugs in the user methods (e.g., stray pointers) frorn corrupting the state of the DBNIS or database and prevents rnalicious methods frorn reading or Inodifying the DBMS state or database as well. Note that the user writing the method need not know that the DBMS is running the method in a separate process: The user code can be linked with a 'wrapper' that turns method invocations and return values into IPCs.

Method Caching

User-defined ADT methods can be very expensive to execute and can account for the bulk of the time spent in processing a query. During query processing, it may 11lake sense to cache the results of methods, in case they are invoked lllultiple times with the same arglunent. Within the scope of a single query, one can avoid calling a Inethod twice on duplicate values in a colurnn by either sorting the table on that colullln or using a hash-based scherne ruuch like that used for aggregation (see Section 14.6). An alternative is to rnaintain a *cache* of rnethod inputs and rnatching outputs as a table in the database. Then, to find the value of a rnethod on particular inputs, we essentially join the input tuples with the cache table. rrhese two approaches can also be cornbined.

Pointer Swizzling

In sorne applications, objects are retrieved into rnernory and accessed frequently through their oids; dereferencin.g rnust be irnplcrnented very efficiently. 801ne systerns rnaintain a ta,hle of oids of objects that are (currently) in InenlOI'.Y. \Vhen an object () is brought into memory, they check each oid contained in O and replace oids of in-memory objects by in-rncrl10ry pointers to those objects. This techrlique, caJled **pointer swizzling**, makes references to in-rnernory objects very .fast. rfhe downside is that when an object is paged out,

23.9 OODBMS

In the introduction of this chapter, we defined an OODBMS as a progranlIning language with support for persistent objects.vVhile this definition reflects the origins of OODBMSs accurately, and to a certain extent the irnplernentation focus of OODBMSs, the fact that OODBl\!ISs support *collection types* (see Section 23.2.1) rnakes it possible to provide a query language over collections. Indeed, a standard has been developed by the Object Database Management Group and is called Object Query Language.

OQL is sirnilar to SQL, with a SELECT----FROM--HWHERE---style syntax (even GROUP BY, HAVING, and ORDER BY are supported) and many of the proposed SQL:1999 extensions. Notably, OQL supports structured types, including sets, bags, arrays, and lists. The OQL treatrnent of collections is rnore uniforlll than SQL:1999 in that it does not give special treatrnent to collections of rows; for exalnple, ()QL allows the aggregate operation COUNT to be applied to a list to C0111pute the length of the list. OQL also supports reference types, path expressions,ADrrs and inheritance, type extents, and SQL-style nested queries. 1'here is also a standard Data Definition Language for OODB1\1Ss (Object Data Language, or ODL) that is sirnilar to the DDL subset of SQL but supports the additional features found in OODBMSs, such as ADT definitions.

23.9.1 The ODMG· Data Model and ODL

'The ODl\iIG data rnodel is the basis for an OODBl\iIS, just like the relational data 1nodel is the basis for an RDBMS. A database contains a collection of objects, which are sirnilar to entities in the ER rnode!. Every object has a unique aid, and a database contains collections of objects with Silllilar properties; such a collection is called a class.

The properties of a class are specified using ()l)L and are of three kinds: attributes, relationships, and rnethod8. _Attributes have an atolnic type or a structured type. ODl.l supports the **set**, bag, **list**, array, and **struct** t,ype constructors; these are just **setof**, bagof, **listof**, ARRAY, and ROW in the terrninology of Section 2:3.2.1.

R,elationships have a type that is either a reference to an object or a collection of such references. A relationship captures how an object is related to one or r1101'e objects of the same class or of a different class. A relationship in the ()IJIvIG- rnodel is really just a binary relationship in the sense of the ER Inodel. A rela.tionship has a corresponding **inverse relationship**; intuitively, it is the relationship 'in the other clirection.' For exarnple, if a Inovie is being

Class == Interface + Implenlentation: Properly speaking, a class consists of an interface together\vith an irnpleluentation of the interface. An ODL interface definition is implemented in an OODBMS by translating it into declarations of the object-oriented language (e.g., C++, Snlalltalk or Java) supported by the OODBMS. If we consider C++, for instance, there is a library of classes that irnplcrnent the ODL constructs. There is also an Object Manipulation Language (OML) specific to the programlning language (in our exanlple, C++), which specifies how database objects are manipulated in the progralnnl.ing .language. rr.he .goal is to seamlessly <u>integrate the prograrnrning language and the database features.</u>

shown at several theaters and each theater shows several rnovies, we have two relationships that are inverses of each other: *shownAt* is associated with the class of movies and is the set of theaters at which the given movie is being shown, and *nowShowing* is associated with the class of theaters and is the set of rnovies being shown at that theater.

Methods are functions that can be applied to objects of the class. There is no analog to methods in the ER or relational models.

The keyword interface is used to define a class. For each interface, we can declare an extent, which is the narne for the current set of objects of that class. The extent is analogous to the instance of a relation and the interface is analogous to the scherna. If the user does not anticipate the need to work with the set of objects of a given class-it is sufficient to manipulate individual objects—the extent declaration can be ornitted.

The following ()DL definitions of the Movie and Theater classes illustrate these concepts. (While these classes bear S(Hne resernblance to the Dinky database scherna, the reader should not look for an exact parallel, since we have rnodified the exarnple to highlight ()DL features.)

```
interface Movie
    (extent Movies key rnovieNarne)
    { attribute date start;
    attribute date end;
    attribute string rnovienarne;
    relationship Set⟨Theater⟩ shownAt inverse Theater::nowSho\ving;
    }
```

The collection of database objects whose class is Movie is called Movies. No two objects in lVIovies have the sarne *rnovieNarne* value, as the key declaration indicates. Each lnovie is shown at a set of theaters and is shown during the specified period. (It \vould be rnore realistic to associate a different period with each theater, since a 1110vie is typically played at different theaters over different periods. While we can define a class that captures this detail, we have chosen a sirnpler definition for our discussion.) A theater is an object of class Theater, defined as:

```
interface Theater
    (extent Theaters key theaterNarne)
    { attribute string theaterName;
    attribute string address;
    attribute integer ticketPrice;
    relationship Set(Movie) nowShowing inverse .lVlovie::shownAt;
    float numshowingO raises(errorConntingMovies);
    }
```

Each theater shows several movies and charges the same ticket price for every movie. Observe that the *shownAt* relationship of Movie and the *nowShowing* relationship of Theater are declared to be inverses of each other. Theater also has a lllethod *numshowing()* that can be applied to a theater object to find the number of movies being shown at that theater.

ODL also allows us to specify inheritance hierarchies, as the following class definition illustrates:

```
interface SpecialShow extends lVlovie
    (extent SpecialShows)
    { attribute integer I11axinnunAttendees;
    attribute string benefitCharity;
    }
```

An object of class SpecialShow is an object of class Movie, with SOI11e additional properties, as discussed in Section 23.5.

23.9.2 OQL

rIhe ODMG query language OQL was deliberately designed to have syntax sirnilar to SQL to rnake it easy for users falniliar with SQL to learn ()QL. Let us begin with a query that finds pairs of Inovies and theaters such that the rnovie is shown at the theater and the theater is showing lHore than one rnovie:

```
SELECT Innarne: M.movieName, tnaIne: T.theaterName
```

```
FROM     lViovies M, lVLsho\vnAt T
WHERE    T.nurllshowing() > 1
```

The SELECT clause indicates how we can give Ilallles to fields in the result: The two result fields are called *rnnarne* and *tname*. The part of this query that differs frorn SQL is the FROM clause. The variable 1V[is bound in turn to each rnovie in the extent Movies. For a given rnovie M, we bind the variable T in turn to each theater in the collection *M.shownAt*. Thus, the use of the path expression *M.shownAt* allows us to easily express a nested query. The following query illustrates the grouping construct in OQL:

```
SELECT    T.ticketPrice,
          avgNurn: AVG(SELECT P.T.nurnshowingO FROM partition P)
FROM      l'heaters T
GROUP BY  T.ticketPrice
```

For each ticket price, we create a group of theaters with that ticket price. This group of theaters is the partition for that ticket price, referred to using the OQL keyword **partition**. In the SELECT clause, for each ticket price, we cornpute the average nunlber of rnovies shown at theaters in the partition for that ticketPrice. OQL supports an interesting variation of the grouping operation that is missing in SQL:

```
SELECT    low, high,
          avgNllln: AVG(SELECT P.T.nurnshowingO FROM partition P)
FROM      Theaters T
GROUP BY  low: T.ticketPrice < 5, high: rr.ticketPrice >= 5
```

The GROUP BY clause now creates just two partitions called *low* and *high*. Each theater object T is placed in one of these partitions based on its ticket price. In the SELECT clause, lo'wand *high* are boolean variables, exactly one of which is **true** in any given output tuple; **partition** is instantiated to the corresponding partition of theater objects. In our exarnple, we get two result tuples. ()ne of thern has *low* equal to **true** and *avgNum* equal to the average nurnber of rnovies shown at theaters \vith a low ticket price. The second tuple has *high* equal to **true** and *avgNum* equal to the average nurnber of Inovies shown at theaters with a high ticket price.

The next query illustrates OQL support for queries that return collections other than set and rnultiset:

```
(SELECT   rr.theaterNarne
FROM      Theaters 'T
ORDER BY  T.ticketPrice DESC) [0:4]
```

The ORDER BY clause makes the result a list of theater naInes ordered by ticket price. The clcrnents of a list can be referred to by position, starting \vith position 0. Therefore, the expression [0:4] extracts a list containing the names of the five theaters \vith the highest ticket prices.

OQL also supports DISTINCT, HAVING, explicit nesting of subqueries, view definitions, and other SQL features.

23.10 COMPARING RDBMS, OODBMS, AND ORDBMS

Now that we have covered the lnain object-oriented DBMS extensions, it is tirne to consider the two lnain variants of object-databases, OODBMSs and ORDBJVISs, and cornpare thern with RDBMSs. Although we presented the concepts underlying object-databases, we still need to define the tenns OODBMS and ORDBMS.

An **ORDBMS** is a relational DBl\1S with the extensions discussed in this chapter. (Not all ORDBMS systerlls support all the extensions in the general forrn that we have discussed theIn, but our concern in this section is the paradigrll itself rather than specific systenls.) An **OODBMS** is a progranlrning language with a type systern that supports the features discussed in this chapter and allows any data object to be **persistent**; that is, to survive across different prograrn executions. Many current systerns conform to neither definition entirely but are lIluch closer to one or the other and can be classified accordingly.

23.10.1 RDBMS versus ORDBMS

COlllparing an RDBMS with an OI{DBMS is straightforward. An R,DBl\1S does not support the extensions discussed in this chapter. rrhe resulting sirnplicity of the data rnodel rnakes it easier to optirnize queries for efficient execution, for example. A relational systern is also easier to use because there are fewer features to master. ()n the other hand, it is less versatile than an ()HJ)BiviS.

23.10.2 OODBMS versus ORDBMS: Similarities

OODBMSs and ()HI)BJVISs both support user-defined ADTs, structured types, ol)ject identity and reference types, and inheritance. Both support a query language for rnanipulating collection types. ORDBMSs support an extended fonn of SQL, and 001)131\18s support ODL/OQL. The similarities are by no rneans accidental ORDBMSs consciously try to add OODBMS features to an RI)B1\118, and OODBMSs in turn have developed query languages based on relational query languages. Both OODBMSs and ORDBMSs provide DBMS functionality such as concurrency control and recovery.

23.10.3 OODBMS versus ORDBMS: Differences

The fundaulental difference is really a philosophy that is carried all the way through: OODBMSs try to add DBMS functionality to a progranllning language, whereas ORDBMSs try to add richer data types to a relational DBlVIS. Although the two kinds of object-databases are converging in terrns of functionality, this difference in their underlying philosophy (and for most systeIns, their irnplementation approach) has iInportant consequences in terIllS of the issues emphasized in the design of these DBJVISs and the efficiency with which various features are supported, as the following comparison indicates:

- OODBMSs airn to achieve seamless integration with a programrning language such as C++, Java, or Smalltalk. Such integration is not an important goal for an ORDBMS. SQL:1999, like SQL-92, allows us to embed SQL commands in a host language, but the interface is very evident to the SQL programer. (SQL:1999 also provides extended prograrnming language constructs of its own, as we saw in Chapter 6.)

- An OODBMS is aimed at applications where an object-centric viewpoint is appropriate; that is, typical user sessions consist of retrieving a few objects and working on theHl for long periods, with related objects (e.g., objects referenced by the original objects) fetched occasionally. Objects rnay be extrelnely large and rnay have to be fetched in pieces; therefore, attention Inust be paid to buffering parts of objects. It is expected that rnost applications can cache the objects they require in rnemory, once the objects are retrieved froIn disk. rrherefore, considerable attention is paid to rnaking references to ill-lnernory objects efficient. Transactions are likely to be of very long duration and holding locks until the end of a transaction Inay lead to poor perfonnance; therefore, alternatives to Two-Phase Locking HUlst be used.

  An ORDBMS is optirnized for applications in which large data collections are the focus, even though objects rnay have rich structure and be fairly large. It is expected that applications will retrieve data frorn disk extensively and optirnizing disk access is still the rnain concern for efficient execution. Transactions are assurned to be relatively short and traditional R.DB1VIS techniques are typically used for concurrency control and recovery.

- The query facilities of OQL are not supported efficiently in rnost OODBMSs, whereas the query facilities are the centerpiece of an ()HI)Bl\1S. To scnne extent, this situation is the result of different concentrations of effort in the clevelopment of these systerns. '1'0 a significant extent, it is also a consequence of the systerlls' being optirnized. for very different kinds of applications.

23.11 REVIEW QUESTIONS

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

- Consider the extended Dinky exarnple frorn Section 23.1. Explain how it lllotivates the need for each of the following object-database features: *'User-defined struct'Ured types, abstract data types (AD Ts), inheritance,* and *object identity.* (Section 23.1)
- What are *structured data types?* What are *collection types,* in particular? Discuss the extent to which these concepts are supported in SQL:1999. What irnportant type constructors are lllissing? What are the limitations on the ROWand ARRAY constructors? (Section 23.2)
- What kinds of operations should be provided for each of the structured data types? To what extent is such support included in SQL:1999? (Section 23.3)
- What is an *abstract data type?* How are nlethods of an abstract data type defined in an external programnling language? (Section 23.4)
- Explain *inheritance* and how new types (called *subtypes)* extend existing types (called *supertypes).* What are *rnethod overloading* and *late b'inding?* What is a *collect'ion hierarchy?* Contrast this with inheritance in prograrn-Ining languages. (Section 23.5)
- How is an *object identifier (aid)* different froln a record id in a relational DElVIS? How is it different froIn a URL? What is a *reference type?* Define *deep* and *shallow* equalit,y and illustrate thern through an exarnple. (Section 23.6)
- 'The rnultitude of data types in an ORDBMS allows us to design a rnore natural and efficient databa"se schema but introduces some new design choices. I)iscuss ORDBMS database design issues and illustrate your discussion using an exalnple application. (Section 23.7)
- Irnplernenting an ORDBMS brings new challenges. The systcrll rnust store large ADTs and structured types that rnight be very large. Efficient and extensible index rnechanisrns lllUSt be provided. Examples of new functionality include *user-defined aggregation functions* (we can define new aggregation functions for our AI)Ts) and *method security* (the systcrIl has to prevent user-defined rnethods fronl cornprornising the security of the DBMS). ExarIlples of new techniques to increase perfonnance include

method caching and *pointer swizzling.* The optirnizer must know about the new functionality and use it appropriately. Illustrate each of these challenges through an exarnple. (Section 23.8)

- Compare OODBIvISs with ORDBMSs. In particular, cornpare OQL and SQL:1999 and discuss the underlying data rnode!. (Sections 23.9 and 23.10)

EXERCISES

Exercise 23.1 Briefly answer the following questions:

1. What are the new kinds of data types supported in object-database systcrIls? Give an exarnple of each and discuss how the exanlple situation would be handled if only an RDBMS were available.
2. What rIlust a user do to define a new ADT?
3. Allowing users to define rIlethods can lead to efficiency gains. Give an exarnple.
4. What is late binding of nlCthods? Give an exarnple of inheritance that illustrates the need for dynamic binding.
5. What are collection hierarchies? Give an exalIlple that illustrates how collection hierarchies facilitate querying.
6. Discuss how a DBMS exploits encapsulation in ilnplernenting support for ADTs.
7. Give an exarnple illustrating the nesting and unnesting operations.
8. Describe two objects that are deep equal but not shallow equal or explain why this is not possible.
9. Describe two objects that are shallow equal but not deep equal or explain why this is not possible.
10. COlnpare RDBNISs with ORDBMSs. Describe an application scenario for which you would choose an RDBMS and explain why. Silnilarly, describe an application scenario for which you would choose an ORDBMS and explain why.

Exercise 23.2 Consider the Dinky schclna shown in Figure 23.1 and all related lncthocls defined in the chapter. Write the following queries in SQL:1999:

1. How luany filrns were shown at theater *tno* = 5 between January 1 and February 1 of 2002'1
2. What is the lowest budget for a filnl with at least two stars?
3. Consider theaters at which a fihu directed by Steven Spielberg started showing on January 1, 2002. For each such theater, print the narnes of all countries within a 100-ruile radius. (You can use the *overlap* and *radius* rnethods illustrated in Figure 2:3.2.)

Exercise 23.3 In a cornpany database, you need to store infornation about clnployees, departrnents, and children of erIlployees. For each ernployec, identified by *ssn*, you rnust record *years* (the number of years that the ernployee has worked for the cornpany), *phone,* and *photo* infornation. There are two subclasses of ernployees: contract and regular. Salary is coruputed by invoking a rnethod that takes *years* as a parameter; this rnethod has a different irnplernentation for each subclass. Further, for each regular employee, you IIlUSt record the name and age of every child. The rnost conunon queries involving children are sirnilar to "Find the average age of Bob's children' and "Print the narnes of all of Bob's children."

A photo is a large image object and call be stored in one of several irnage fonnats (e.g., gif, jpeg). You want to define a *display* rnethod for image objects; display IIlust be defined differently for each irnage fonnat. 'For each department, identified by *dno*, you rnust record *dname*, *budget*, and *workers* infonnation. *Workers* is the set of crllployees who work in a given department. Typical querie.s involving workers include, "Find the average salary of all workers (across all departments)."

1. Using extended SQL, design an ORDBIVIS scherna for the cornpany database. Show all type definitions, including rnethod definitions.
2. If you have to store this infonnation in an RDBMS, what is the best possible design?
3. Cornpare the ORDBMS and RDBMS designs.
4. If you are told that a COlllInon request is to display the irnages of all employees in a given departruent, how would you use this inforulation for physical database design?
5. If you are told that an ernployee's ilnage rnust be displayed whenever any information about the employee is retrieved, would this affect your scherna design?
6. If you are told that a COU1Inon query is to find all erTlployees who look sirnilar to a given image and given code that lets you create an index over all irnages to support retrieval of sinlilar images, what would you do to utilize this code in an OR.DBMS?

Exercise 23.4 ORDBMSs need to support efficient access over collection hierarchies. Consider the collection hierarchy of Theaters and Theater_cafes presented in the Dinky exanlple. In your role as a DBMS illlplernentor (not a DBA), you rnust evaluate three storage alternatives for these tuples:

- All tuples for all kinds of theaters are stored together all disk in an arbitrary order.
- All tuples for all kinds of theaters are stored together on disk, with the tuples that are frOIll TheateLcafes stored directly after the last of the non-cafe tuples.
- T'uples froIll Theater_cafes are stored separately froIll the rest of the (non-cafe) theater tuples.

1. F'or each storage option, describe a rnechanisrn for distinguishing plain theater tuples frorn Theater_cafe tuples.
2. For each storage option, describe hmv to handle the insertion of a new non-cafe tuple.
3. Which storage option is 1110St efficient for queries over all theaters? Over just TheateLcafes? In terrns of the nurnber of 1/Os, how rnuch rnore efficient is the best technique for each type of query cornpared to the other two techniques?

Exercise 23.5 Different ORDBMSs use different techniques for building indexes to evaluate queries over collection hierarchies. For our Dinky example, to index theaters by name there are two COIllIIlon options:

- Build one 13+ tree index over Theaters.*name* and another 13+ tree index over Theater_cafes.*narne.*
- Build one B+ tree index over the union of Theaters.*name* and Theater_cafes.*name.*

1. Describe how to efficiently evaluate the following query using each indexing option (this query is over aU kinds of theater tuples):

    SELECT * FROM Theaters T WHERE T.narne= 'Majestic'

    Give an estimate of the nurnber of l/Os required in the two different scenarios, assurning there are 1 rnillion standard theaters and 1000 theater-cafes. Which option is Inore efficient?

2. Perforrn the saIlle analysis over the following query:

    SELECT * FROM Theater-cafes 'I' WHERE T.nalne = 'Majestic'

3. For clustered indexes, does the choice of indexing technique interact with the choice of storage options? For unclustered indexes?

Exercise 23.6 Consider the following query:

```
SELECT thurnbnail(Lirnage)
FROM   lInages I
```

Given that the 1.*image* colurnn 111ay contain duplicate values, describe how to use hashing to avoid conlputing the *thum,bnail* function rnore than once per distinct value in processing this query.

Exercise 23.7 You are given a two-dimensional, $n \times n$ array of objects. Assume that you can fit 100 objects on a disk page. Describe a way to layout (chunk) the array onto pages so that retrievals of square m x m subregions of the array are efficient. (Different queries request subregions of different sizes, i.e., different m values, and your arrangement of the array onto pages should provide good perforrnance, on average, for all such queries.)

Exercise 23.8 An ORDBMS optiruizer is given a single-table query with n expensive selection conditions, $\sigma_n(\ldots(\sigma_1(T)))$. For each condition σ_i, the optirnizer can estinlate the cost c_i of evaluating the condition on a tuple and the reduction factor of the condition r_i. Assurne that there are t tuples in T.

1. How many tuples appear in the output of this query?
2. Assurning that the query is evaluated as shown (without reordering selections), what is the total cost of the query? Be sure to include the cost of scanning the table and applying the selections.
3. In Section 23.8.2, it was asserted that the optiruizer should reorder selections so that they are applied to the table ill order of increasing rank, where $\text{rank}_i = (r_i - 1)/c_i$. Prove that this assertion is optirual. That is, show that no other ordering could result in a query of lower cost. (Hint: It may be easiest to consider the special case where $n = 2$ first and generalize from there.)

Exercise 23.9 ORDBIVlSs support references as a data type. It is often clailnecl that using references instead of key-foreign key relationships will give rnuch higher perfonnance for joins. 'This question asks you to explore this issue.

- Consider the following SQL:1999 DDL which only uses straight relational constructs:

```
CREATE TABLE R(rkey integer, rdata text);
CREATE TABLE S(skey integer, rfkey integer);
```

Assurne that we have the following straightforward join query:

```
SELECT  S.skey, H..relata
FROM    S, R
WHERE   S.rfkey = R.rkey
```

■ Now consider the following SQL:1999 ORDBMS schelna:

```
CREATE TYPE r_t AS ROW(rkey integer, rdata text);
CREATE TABLE R OF r_t REF is SYSTEM GENERATED;
CREATE TABLE S (skey integer, r REF(r_t) SCOPE R);
```

Assurne we have the following query:

```
SELECT  S.skey, S.r.rkey
FROM    S
```

What algorithrll would you suggest to evaluate the pointer join in the ORDBMS scherna? How do you think it will perform versus a relational join on the previous scherna?

Exercise 23.1.0 Many object-relational systerns support set-valued attributes using some variant of the setof constructor. For eXaInple, assurning we have a type `person_t`, we could have created the table Filrns in the Dinky Schema in Figure 23.1 as follows:

CREATE TABLE *Films(filrnno* integer, *title* `text`, *stars* setof Person);

1. Describe two ways of irnpleIllenting set-valued attributes. One way requires variable-length records, even if the set elements are all fixed-length.
2. Discuss the irnpact of the two strategies on optimizing queries with set-valued attributes.
3. Suppose you would like to create an index on the column stars in order to look up filrns by the narne of the star that has starred in the filIIl. For both irnplenlentation strategies, discuss alternative index structures that could help speed up this query.
4. What types of statistics should the query optirnizer rnaintain for set-valued attributes? How do we obtain these statistics'?

BIBLIOGRAPHIC NOTES

A nurnber of the object-oriented features described here are based in part on fairly old ideas in the prograrnrning languages cornrnunity. [42] provides a good overview of these ideas in a database context. Stonebraker's book [719J describes the vision of ORDBMSs ernbodied by his company's early product, Illustra (now a product of Inforrnix). Current connnercial DBMSs with object-relational support include Infonnix Universal Server, IBM D13/2 CS V2, and UniSQL. An new version of Oracle is scheduled to include ORDBMS features as well.

Many of the idcas in current object-relational systerlls carne out of a few prototypes built in the 1980s, especially POSTGRES [723], Starburst [351], and 02 [218].

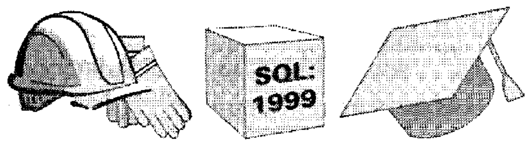
The iclea of an object-oriented database was first articulated in [197], \vhich described the GernStone prototype system. Other prototypes includeDASDBS [657], EXODUS [130], nus [273], Ol:>jectStore [463], ODE, [18] ORION [432], SHORE [129], and THOR [482]. 02 is actually an early example of a systenl that was beginning to rnerge the thcrnes of ORDBrvISs and OODBMSs—it could fit in this list as well. [41] lists a collectioll of features that are generally considered to belong in an OODBMS. Current cornrnercially available OODBMSs include GelllStone, Itasca, 02, Objectivity, ObjectStore, Ontos, Poet, and Versant. [431] cornpares OODBIvISs and RDBMSs.

Database support for ADTs was first explored in the INGRES and POSTGRES projects at U.C. Berkeley. The basic ideas are described in [716], including Inechanisllls for query processing and optilnization with ADTs as well as extensible indexing. Support for ADTs was also investigated in the Dannstadt database systern, [480]. Using the POSTGRES index extensibility correctly required intimate knowledge of DBTvIS-internal transaction ruechanisllls. Generalized search trees were proposed to solve this problern; they are described in (376], with concurrency and ARIES-based recovery details presented in [447]. [672] proposes that users lnust be allowed to define operators over ADT objects and properties of these operators that can be utilized for query optiInization, rather than just a collection of lnethods.

Array chunking is described in (653]. Techniques for luethod caching and optimizing queries with expensive lnethods are presented in [37:3, 165]. Client-side data caching in a client-server 00D131\I1S is studied in [283]. Clustering of objects on disk is studied in [741]. Work on nested relations was an early precursor of recent research on complex objects in OODBMSs and ORD13IvISs. One of the first nested relation proposals is (504]. MVDs play an inlportant role in reasoning about reduncancy in nested relations; see, for exalnple, [579]. Storage structures for nested relations were studied in (215].

Fonnal rnodels and query languages for object-oriented databases have been widely studied; papers include [4, 56, 75, 125, 391, 392, 428, 578, 724]. [427] proposes SQL extensions for querying object-oriented databases. An early and elegant extension of SQL with path expressions and inheritance was developed in GEM [791]. There has been ITluch interest in cornbining deductive and object-oriented features. Papers in this area include (44, 288, 495, 556, 706, 793]. See [3] for a thorough textbook discussion of fonnal aspects of object-orientation and query languages.

[433, 435, 721, 796] include papers on DBMSs that would now be tenned object-relational and/or object-oriented. [794] contains a detailed overview of scherna and database evolution in object-oriented database systenls. A thorough presentation of SQL:1999 can be found in [525), and advanced features, including the object extensions, are covered in [523]. A short survey of new SQL:1999 features can be found in [2:37]. The incorporation of several SQL:1999 features into I131v1 D132 is described in [128J. OQL is described in [141]. It is based to a large extent on the 02 query language, which is described, together with other aspects of 02, in the collection of papers [55].



24

DEDUCTIVE DATABASES

- What is the nlotivation for extending SQL with recursive queries?
- What important properties must recursive programs satisfy to be practical?
- What are least lnodels and least fixpoints and how do they provide a theoretical foundation for recursive queries?
- What cOlnplications are introduced by negation and aggregate operations? How are they addressed?
- What are the challenges in efficient evaluation of recursive queries?
- **Key concepts:** Datalog, deductive databases, recursion, rules, infeI'ences, safety, range-restriction; least model, declarative semantics; least fixpoint, operational semantics, fixpoint operator; negation, stratified program.s; aggregate operators, rnultiset generation, grouping; efficient evaluation, avoiding repeated inferences, Seminaive fixpoint evaluation; pushing query selections, lVlagic Sets rewriting

For 'Is' and 'Is-Not' though with Rule and Line,
And 'Up-and-Down' by Logic I define,
Of all that one should care to fathorn, I
Was never deep in anything but-------\Vine.

.. ·*Rubaiyat of Omar !(hayyarn,* Translated by Edward Fitzgerald

Relational database rnanagernent systenls have been enonnously successful for C),chninistrative da,ta processing. In recent years, however, as people have tried to use database systerus in increasingly cornplex applications, some irnportant linl-itations of these systellls have been exposed. For sonle applications, the query language and constraint definition capabilities have been found inadequate. As an exarnple, sonle cornpanies ulaintain a huge parts inventory database and frequently want to ask questions such as, "Are we running Iowan any parts needed to build a ZX600 sports car?" or "What is the total cornponent and assernbly cost to build a ZX600 at today's part prices?" These queries cannot be expressed in SQL-92.

We begin this chapter by discussing queries that cannot be expressed in relational algebra or SQL and present a rnore powerful relational language called *Datalog.* Queries and views in SQL can be understood as **if-then** rules: "**If** some tuples exist in tables mentioned in the FROM clause that satisfy the conditions listed in the WHERE clause, **then** the tuple described in the SELECT clause is included in the answer." Datalog definitions retain this if-then reading, with the significant new feature that definitions can be *recursive,* that is, a table can be defined in terms of itself. The SQL:1999 standard, the successor to the SQL-92 standard, requires support for recursive queries, and a large subset S0111e systerlls, notably IBM's DB2 DBMS, already support thelu.

Evaluating Datalog queries poses some additional challenges, beyond those encountered in evaluating relational algebra queries, and we discuss sonle iUlportant ilnplernentation and optimization techniques developed to address these challenges. Interestingly, some of these techniques have been found to irnprove perforrnance of even nonrecursive SQL queries and have therefore been implernented in several current relational DBMS products.

In Section 24.1, we introduce recursive queries and Datalog notation through an exaruple. We present the theoretical foundations for recursive queries, least fixpoints and least rnodels, in Section 24.2. We discuss queries that involve the use of negation or set-difference in Section 24.3. Finally, we consider techniques for evaluating recursive queries efficiently in Section 24.5.

24.1 INTR()DUCTION TO RECURSIVE QUERIES

We begin with a sinlple example that illustrates the li111its of SQL-92 queries and the power of recursive definitions. Let Assernbly be a relation \vith three fields *part*, *subpart*, and *qty*. An exaınple instance of Assernbly is shown in Figure 24.1. Each tuple in Assernbly indicates IH}w Inany copies of a particular subpart are COlltained in a given part. The first tuple indicates, for example, that a trike contains three wheels. The Assclnbly relation can be visuaJized as a tree, as sho\vn in Figure 24.2. A. tuple is shown as an edge going frorn the part to the subpart, with the *qty* value as the edge label.

part	*subpart*	*qty*
trike	\vheel	3
trike	fralne	1
frarne	seat	1
franle	pedal	1
wheel	spoke	2
\vheel	tire	1
tire	run	1
tire	tube	1

Figure 24.1 An Instance of Assembly

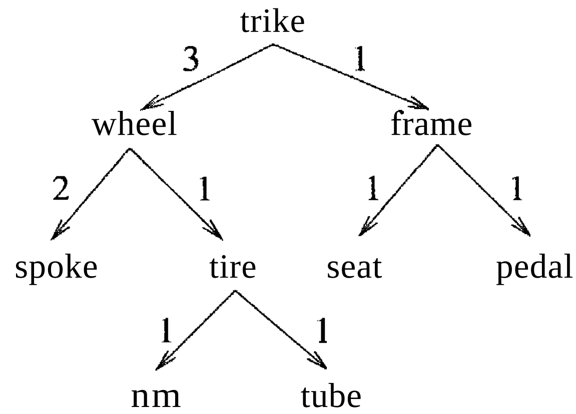


Figure 24.2 Assembly Instance Seen as a Tree

A natural question to ask is, "What are the cornponents of a trike?" Rather surprisingly, this query is inlpossible to write in SQL-92. Of course, if we look at a given instance of the Assernbly relation, we can write a 'query' that takes the union of the parts that are used in a trike. But such a query is not interesting---we want a query that identifies all components of a trike for *any* instance of Assembly, and such a query cannot be written in relational algebra or in SQL-92. Intuitively, the problem is that we are forced to join the Asselnbly relation with itself to recognize that *trike* contains *spoke* and *tire,* that is, to go one level down the Assenlbly tree. For each additional level, we need an additional join; two joins are needed to recognize that *trike* contains *rim,* which is a subpart of *tire.* Thus, the ntullber of joins needed to identify all subparts of *trike* depends on the height of the Assen1bly tree, that is, on the given instance of the Assembly relation. No relational algebra query works for all instances; given any query, we can construct an instance whose height is greater than the nurnber of joins in the query.

24.1.1 Datalog

We now define a relation called Cornponents that identifies the cOlnponents of every part. Consider the following program, or collection of rules:

Components (Part, SUbpart) "- Assembly (Part, SUbpart, Qty) "
Components (Part, Subpart) .- Assembly (Part, Part2, Qty) ,
Components (Part2, Subpart)"

These are rules in Datalog, a relational query language inspired by Prolog, the well-known logic progranuning language; indeed, the notation follows Prolog. The first rule should be read as follo\vs:

For all values of Part, Subpart, and Qty,

> if there is a tuple (Part, Subpart, Qty⟩ in Assclnbly,
> then there IJIUSt be a tuple ⟨Part, Subpart) in (;olnponents.

The second rule should be read as follows:

> For all values of Part, Part2, Subpart, and Qty,
> if there is a tuple (Part, Part2, Qty⟩ in Assernbly and
> a tuple ⟨Part2, Subpart) in Components,
> then there HUlst be a tuple ⟨Part, Subpart) in C()lnponents.

The part to the right of the :- sYlnbol is called the **body** of the rule, and the part to the left is called the **head** of the rule. The syrnbol :- denotes logical irnplication; if the tuples lIlentioned in the body exist in the database, it is irnplied that the tuple rnentioned in the head of the rule rnust also be in the database. (Note that the body could be ernpty; in this case, the tuple rnentioned in the head of the rule rnust be included in the database.) 1'herefore, if we are given a set of Assenlbly and Cornponents tuples, each rule can be used to infer, or **deduce**, sorne new tuples that belong in COlnponents. This is why database systerns that support Datalog rules are often called **deductive database systems**.

By assigning constants to the variables that appear in a rule, we can infer a specific Coruponents tuple. For example, by setting Part=*trike*, Subpart=*wheel*, and Qty=*3*, we can infer that ⟨*trike*, *wheel)* is in eoulponents. Each rule is really a *ternplate* for Inaking **inferences**: An inference is the use of a rule to generate a new tuple (for the relation in the head of the rule) by substituting constants for varia,bles in such a way that every tuple in the rule body (after the substitution) is in the corresponding relation instance.

By considering each tuple in Asselnbly in turn, the first rule allows us to infer that the set of tuples obtained by taking the projection of Assernbly onto its first two fields is in CCHnponents.

The seco11d rule then allo\vs us to cOlnbine previously discovered Cornponents tuples with Assernbly tuples to infer new Cornponents tuples. We can apply the second rule by considering the cross-product of Assernbly and (the current instance of) Cornponents and assigning values to the variables in the rule for each row of the cross-product, one row at a time. ()bserve how the repeated use of the varial)le Part2 prevents certain rows of the cross-product fronl contributing any new tuples; in effect, it specifies an equality join condition on AssenIbly and Cornpouents. The tuples obtained by one application of this rule are shown in Figure 24.3. (In addition, COlnponents contains the tuples obtained by applying the first rule; these are not shown.)

part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rhn
wheel	tube

Figure 24.3 Components Tuples Obtained by Applying the Second Rule Once

part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	run
whe81-	tube
trike	rirn
trike	tube

Figure 24.4 Components Tuples Obtained by Applying the Second Rule Twice

The tuples obtained by a second application of this rule are shown in Figure 24.4. Note that each tuple shown in Figure 24.3 is reinferred. Only the last two tuples are new.

Applying the second rule a third time does not generate additional tuples. rrhe set of Components tuples shown in Figure 24.4 includes all the tuples that can be inferred using the two Datalog rules defining Cornponents and the given instance of Assembly. rrhe components of a trike can now be obtained by selecting all Cornponents tuples with the value *trike* in the first field.

Each application of a Datalog rule can be understood in ternlS of relational algebra. The first rule in our exarnple program simply applies projection to the Assernbly relation and adds the resulting tuples to the Cornponents relation, which is initially ernpty. The second rule joins Assernbly with COlllponents and then does a projection. The result of each rule application is cornbined with the existing set of Cornponents tuples using union.

The only Datalog operation that goes beyond relational algebra is the *repeated* application of the rules defining CCHnponents until no new tuples are generated. This repeated application of a set of rules is called the *fixpoint* operation, and we develop this idea further in the next section.

We conclude this section by rewriting the Datalog definition of Cornponents using SQL:1999 syntax:

```
WITH RECURSIVE Cornponents(Part, Subpart) AS
        (SELECT A1.Part, A1.Subpart FROM Assernbly A1)
        UNION
        (SELECT A2.Part, Cl.Subpart
        FROM    Assernbly A2, Cornponents C1
```

```
        WHERE  A2.Subpart = C1.Part)

SELECT * FROM COlllponents C2
```

The WITH clause introduces a relation that is part of a query definition; this relation is shnilar to a view, but the scope of a relation introduced using WITH is local to the query definition. The RECURSIVE key\vord signals that the table (in our example, Cornponents) is recursively defined. The structure of the definition closely parallels the Datalog rules. Incidentally, if we wanted to find the cornponents of a particular part, for exanlple, *tTikc,* we can sirnply replace the last line with the following:

```
SELECT * FROM Cornponents C2
WHERE  C2.Part = 'trike'
```

24.2 THEORETICAL FOUNDATIONS

We classify the relations in a Datalog progialn as either output relations or input relations. **Output relations** are defined by rules (e.g., COluponents), and **input relations** have a set of tuples explicitly listed (e.g., Assembly). Given instances of the input relations, we Inust compute instances for the output relations. The meaning of a Datalog prograrIl is usually defined in two different ways, both of which essentially describe the relation instances for the output relations. Technically, a **query** is a selection over one of the output relations (e.g., all Components tuples C with C.*paTt* = *tTike).* However, the lueaning of a query is clear once we understand how relation instances are associated with the output relations in a Datalog progranl.

rrhe first approach to defining the sernantics of a Datalog prograll, called the *least model semantics*, gives users a way to understand the prograrn without thinking about how the prograrn is to be executed. That is, the sernanties is *declarative*, like the sernantics of relational calculus, and not *operational* like relational algebra sClnantics. This is irnportant becC111se recursive rules lnake it difficult to understand a· prograIll in tcrrns of an evaluation strategy.

The second approach, called the *least fixpoint 8crnantic8,* gives a conceptual evaluation strategy to COlnpute the desired relation insta.nces. This serves as the basis for recursive query evaluation in a DBMS. More efficient evaluation strategies are used in an actual iInplernentation, but their correctness is sho\vI1 by demonstrating their equivalence to the least fixpoint approach. rrhe fixpoint sClnantics is thus operational and. plays a role analogous to that of relational algebra sernalltics for nonrecursive queries.

24.2.1 Least Model Semantics

We want users to be able to understand a Datalog progTarn by understanding each rule independent of other rules, with the lneaning: *If the body is true, the head is also true.* This intuitive reading of a rule suggests that, given certain relation instances for the relation naines that appear in the body of a rule, the relation instance for the relation rnentioned in the head of the rule 111USt contain a certain set of tuples. If a relation Harne R appears in the heads of several rules, the relation instance for R IIIUSt satisfy the intuitive reading of all these rules. However, we do not want tuples to be included in the instance for R unless they are necessary to satisfy one of the rules defining R. That is, we want to cornpute only tuples for R that are supported by SaIne rule for R.

To lnake these ideas precise, we need to introduce the concepts of rnodels and least models. A model is a collection of relation instances, one instance for each relation in the prograrn, that satisfies the following condition. For every rule in the prograrll, whenever we replace each variable in the rule by a corresponding constant, the following holds:

> *If* every tuple in the body (obtained by our replaceUlent of variables with constants) is in the corresponding relation instance,
>
> *Then* the tuple generated for the head (by the assignrnent of constants to variables that appear in the head) is also in the corresponding relation instance.

Observe that the instances for the input relations are given, and the definition of a rnodel essentially restricts the instances for the output relations.

Consider the rule

```
Components(Part, Subpart) '-  Assembly(Part, Part2, Qty),
                              Components(Part2, Subpart).
```

Suppose we replace the variable Part by the constant *wheel,* Part2 by *tire,* Qty by 1, and Subpart by *rim:*

```
Components(wheel, rim) '- Assembly(wheel, tire, 1),
                          Components(tire, rim).
```

Let A be an instance of Assernbly and C be an instance of COlnpouents. If A contains the tuple ⟨*wheel*, *tire*, 1) and C contains the tuple ⟨*tire*, *rim*⟩, then C rrulst also contain the tuple ⟨*wheel*, *rim*⟩ for the pair of instances A and C to be a rnodel. ()f course, the instances A and C rnust satisfy the inclusion requirenlent just illustrated for *every* assignruent of constants to the variables in the rule: If the tuples in the rule body are in.A. and C, the tuple in the head Inus1; be in C.

As an exarnple, the instances of Asscrnbly shown in Figure 24.1 and Cornpo-nents shown in F'igure 24.4 together fornl a rnodel for the Conlponcnts prograll1.

C;iven the instance of Assernbly shown in Figure 24.1, there is no justification for including the tuple ⟨*spoke*, *pedal)* to the COlnponents instance. Indeed, if we add this tuple to the cornponents instance in Figure 24.4, we no longer have a lllodel for our program, as the following instance of the recursive rule derllonstrates, since *(wheel, pedal)* is not in the Cornponents instance:

```
Components(wheel, pedal) :-  Assembly(wheel, spoke, 2),
                             Components(spoke, pedal).
```

However, by also adding the tuple ⟨*wheel*, *pedal)* to the Cornponents instance, we obtain another rnodel of the Components prograrll. Intuitively, this is un-satisfactory since there is no justification for adding the tuple ⟨*spoke*, *pedal)* in the first place, given the tuples in the Assembly instance and the rules in the prograln.

We address this problern by using the concept of a least rllodel. A least model of a prograrn is a rnodel M such that for every other model M2 of the sarne progranl, for each relation R in the program, the instance for R in]\II is contained in the instance of R in 1\12. The 1nodel forIned by the instances of Assernbly and COlnponents shown in Figures 24.1 and 24.4 is the least rHodel for the CC)lnponents prograllI with the given Assernbly instance.

24.2.2 The Fixpoint Operator

A fixpoint of a function *f* is a value *v* such that the function applied to the value returns the saIlle value, that is, $f(v) = v$. Consider a function applied to a set of values that also returns a set of values. For example, we carl define *double* to be a function tllat Illuitiplies every element of the input set by two and *double+* tobe *double* ∪ *identity.* Thus, *double*({1,2,5}) = {2,4,lO}, and *double+*({1,2,5}) = {1,2,4.,5,lO}.The set of all even integers which happens to be an infinite set-is a fixpoint of the function *double+*. Another fixpoint of the function *double+* is the set of all integers. The first fixpoint (the set of all even integers) is ***smaller*** than the second fixpoint (the set of all integers) because it is contained in the latter.

The least fixpoint of a function is the fixpoint that is slnaller than every other fixpoint of that function. In general, it is not guaranteed that a function has a least fixpoint. For example, there lnay be two fixpoints, neither of \vhich is 81na11er than the other. (Does *double* have a least fixpoint? What is it?)

No\V let us turn to functions over sets of tuples, in particular, functions defined using relational algebra expressions. The Cornponents relation can be defined by an equation of the fonn

$$Components \; = \; \pi_{1,5}(Assembly \bowtie_{2=1} Components) \; \cup \; \pi_{1,2}(Assernbly)$$

1'his equation has the forn1

$$Cornponents \; = \; f(Cornponents, Assembly)$$

where the function f is defined using a relational aJgebra expression. For a given instance of the input relation Assernbly, this can be sirnplified to

$$Components \; = \; f(Co1nponents)$$

The least fixpoint of f is an instance of Cornponents that satisfies this equation. Clearly the projection of the first two fields of the tuples in the given instance of the input relation Assernbly rnust be included in the (instance that is the) least fixpoint of Cornponents. In addition, any tuple obtained by joining Components with Assernbly and projecting the appropriate fields IllUst also be in Components.

A little thought shows that the instance of Components that is the least fixpoint of f can be ccnnputed using repeated applications of the Datalog rules sho\vn in the previous section. Indeed, applying the two Datalog rules is identical to evaluating the relational expression used in defining COlnponcnts. If an application generates Cornponents tuples that are not in the current instance of the Cornponents relation, the current instance cannot be the fixpoint. Therefore, we add the new tuples to Cornponents and evaluate the relational expression (equivalently, the two Datalog rules) again. T'his process is repeated until every tuple generated is already in the current instance of Cornponents. When applying the rules to i,he currerlt set of tuples does not produce any new tuples, we have reached a fixpoint. If CC)lnponents is initialized to the erupty set of tuples. intuitively we infer only tuples that (1,1'e necessary by the definition of a fixpoint, and the fixpoint cornputed is the least fixpoint.

24.2.3 Safe Datalog Programs

Consider the follovving p1'ograrn:

```
ComplexYarts(Part) :- Assembly(Part, Subpart, Qty), Qty > 2.
```

According to this rule, a cOlnplex part is defined to be any part that has Inore than t\vo copies of anyone subpart. For each part lnentioned in the Asselnbly relatioIl, we can easily check whether it is a cOll1plex part. In contrast, consider the following prograrn:

```
PriceYarts(Part, Price) '-
                Assembly(Part, Subpart, Qty), Qty> 2.
```

This variation seeks to associate a price with each cornplex part. IIowever, the variable *Price* does not appear in the body of the rule. This Ineans that an infinite number of tuples must be included in any model of this progralll. To see this, suppose we replace the variable Part by the constant *trike,* SubPart by *wheel,* and Qty by 3. This gives us a version of the rule with the only remaining variable being *Price:*

```
PriceYarts(trike,Price) :- Assembly(trike, wheel, 3), 3 > 2.
```

Now, any assignment of a constant to *Price* gives us a tuple to be included in the output relation Price_.Parts. For example, replacing *Price* by 100 gives us the tuple Price_Parts(trike,lOO). If the least Inodel of a prograllI is not finite, for even one instance of its input relations, then we say the program is unsafe.

Database systems disallow unsafe programs by requiring that every variable in the head of a rule also appear in the body. Such progralns are said to be range-restricted, and every range-restricted Datalog prograln has a finite least model if the input relation instances are finite. In the rest of this chapter, we assume that prograrns are range-restricted.

24.2.4 Least Model =Least Fixpoint

Does a Datalog prograln always have a least rnodel? ()r is it possible that there are two rnodels, neither of which is contained in the other'? Sirnilarly, does every Datalog progranl have a least fixpoint? VvThat is the relationship between the least rnodel and the least fixpoint of a Datalog prograln?

As we noted earlier, not every function has a least fixpoint. Fortunately, every function defined in terms of relational algebra expressions tl1at do not contain set-difference is ,guaranteed to have a least fixpoint, and the least fixpoint can be cornputed by repeatedly evaluating the functic)ll. This tells us that every l)atalog prograrn has a least fixpoint and that it can l)e cOlnputed by repeatedly applyi11g the rules of the l)rogranl on the given instances of the input relations.

Further, every Datalog program is guaranteed to have a least rnodel and the least rnodel is equal to the least fixpoint of the I>l'ograln. These results (whose proofs we do not discuss) provide the basis for Datalog query processing. 'Users can understand a progTarn in terms of 'If the body is true, the head is also true,' thanks to the least l1lodel sClnantics. The DBMS can COlllpute the answer by repeatedly applying the prograrn rules, thanks to the least fixpoint sernantics and the fact that the least nlodel and the least fixpoint are identical.

24.3 RECURSIVE QUERIES WITH NEGATION

Unfortunately, once set-difference is allo\ved in the body of a rule, there r11ay be no least rnodel or least fixpoint for a program. Consider the following rules:

```
Big(Part):-     Assembly(Part, Subpart, Qty), Qty> 2,
                NOT Small(Part).
Small(Part) :-  Assembly(Part, Subpart, Qty), NOT Big(Part).
```

These two rules can be thought of as an attenlpt to divide parts (those that are mentioned in the first colulnn of the Asselubly table) into two classes, Big and Small. The first rule defines Big to be the set of parts that use at least three copies of some subpart and are not classified as small parts. The second rule defines Small as the set of parts not classified as big parts.

If we apply these rules to the instance of Assembly shown in Figure 24.1, *trike* is the only part that uses at least three copies of senne subpart. Should the tuple *(trike)* be in Big or SUlall? If we apply the first rule and then the second rule, this tuple is in Big. To apply the first rule, we consider the tuples in Asselubly, choose those with Qty > 2 (which is just *(trike))*, discard those in the current instance of Srnal1 (both Big and Small are initially elnpty), and add the tuples that are left to Big. 1'herefore, an application of the first rule adds *(trike)* to Big. Proceeding siInilarly, we can see that if the second rule is applied before the first, $\langle trike \rangle$ is added to Srnall instead of Big.

This program has two fixpoints, neither of 'which is srnaller than the other, as shown in Figure 24.5. (rhe first fixpoint has a Big tuple that does not appear in the second fixpoint; therefore, it is not smaller than the second fixpoint. The second fixpoint has a 81na11 tuple that does not appear in the first fixpoint; therefore, it is D.ot sr11a11e1' than the first fixpoint. The order ill \vhich we apply the rules detennines \vhich fixpoint is cOlnputed; this situation is very unsatisfactory.\Ve want users to be able to understand their queries without thinking (1)out exactly ho\v the evaluation proceeds.

The root of the problerH is the use of NOT. When we apply the first rule, senne irlferences are disallowed because of the presence of tuples in 8ma11. Parts

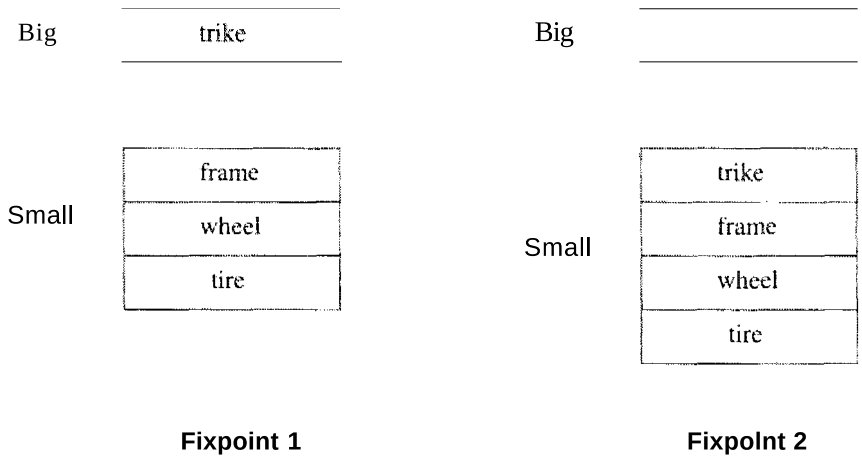


Figure 24.5 Two Fixpoints for the Big/Small Program

that satisfy the other conditions in the body of the rule are candidates for addition to Big; we remove the parts in 8ma11 frorn this set of candidates. Thus, sorne inferences that are possible if 8ruall is ernpty (as it is before the second rule is applied) are disallowed if SInall contains tuples (generated by applying the second rule before the first rule). Here is the difficulty: If NOT is used, the addition of tuples to a relation can *disallow* the inference of other tuples. Without NOT, this situation can never arise; the addition of tuples to a relation can *never* disallow the inference of other tuples.

Range-Restriction and Negation

If rules are allowed to contain NOT in the body, the definition of range-restriction rnust be extended ensure that all range-restricted prograrJlS are safe. If a relation appears in the body of a rule preceded by NOT, we call this a **negated occurrence**. Relation occurrences in the body that are not negated are called **positive occurrences**. A prograrn is **range-restricted** if every variable in the head of the rule appears in sorne positive relation occurrence in the body.

24.3.1 Stratification

A widely used solution to the problern caused by negation, or the use of NOT, is to irnpose certain syntactic restrictions on prograrlls. rrhese restrictions can be easily checked and programs that satisfy them have a natural meaning.

We say that a table$T$ **depends on** a table S if sorne rule with T in the head contains S, or (recursively) contains a predicate that depends on S, in the bod:y. A recursively defined predicate always depends on itself. For example, Big depends on Sruall (and on itself). Indeed, the tables Big and Srnall (l,re nlutually recursive, that is, the definition of 'Big depends on SrnaU and vice versa. We say that a table T **depends negatively** on a table S if some rule with 'T in the head contains NOT S, or (recursively) contains a predicate that depends negatively on S, in the body.

Suppose we classify the tables in a prograrll into **strata** or layers as follows. The tables that do not depend on any other tables are in straturll 0. In our Big/S1nall exarnple, ASSCIIlbly is the only table in stratu1Il 0. Next, we identify tables in straturll 1; these are tables that depend only on tables in stratuln 0 or straturn 1 and depend negatively only on tables in straturn 0. Higher strata are sirnilarly defined: The tables in **straturni** are those that do not belong to lower strata, depend only on tables in stratuIll i o[lower strata, and depend negatively only on tables in lower strata. A **stratified program** is one whose tables can be classified into strata according to the above algoritlull.

rrhe Big/Sruall progralIl is not stratified. Since Big and Snlall depend on each other, they 1nust be in the sarne straturn. Ho\vever, they depend negatively on each other, violating the requirc1Ilent that a table can depend negatively only on tables in lower strata. Consider the following variant of the Big/Srnall progra1Il, in which the first rule has been rnodified:

```
Big2(Part)  :- Assembly(Part, Subpart, Qty), Qty> 2.
Smal12(Part)  :- Assembly(Part, Subpart, Qty), NOT Big2(Part).
```

This prograrn is stratified. Slnall2 depends on Big2 but Big2 does not depend on 8111a1l2. Assernbly is in stratu111 0, Big is in straturn 1, and Srna1l2 is in straturn 2.

A stratified prograrn is evaluated stratu1n-by-straturn, starting with stratunl 0. 'To evaluate a straturn, we cornpllte the fixpoint of all rules defining tables in this straturn. When evaluating a straturn, any occurrence of NOT involves a table frorH a lower straturn, which has therefore been corupletely evaluated by now. The tuples in the negated table still disallow sorne inferences, but the effect is cornpletely deterrninistic, given the straturn-by-straturn evaluation. In the example, Big2 is C01uput8(1 before 81na1l2 because it is in a lower straturIl than 8rna112: $\langle trike \rangle$ is added to Big2. Next, 'when we cornpute 81na112, we recognize that $\langle trike \rangle$ is not in 8rna112 l)ecause it is already in Big2.

Incidentally, note that the stratified Big/Srnall progranl is not even recursive. If we replace Assernbl.y by Cornponents, we obtain a recursive, stratified prograrn: }\,sscrnbly is in straturn 0, Cornponents is in stratlull 1, Big2 is also in stratum 1, and 81na112 is in straturn 2.

Intuition behind Stratification

Consider the stratified version of the Big/Slnall prograrll. The rule defining Big2 forces us to add $\langle trike \rangle$ to Big2 and it is natural to assume that $\langle trike \rangle$ is the only tuple in Big2, because we have no supporting evidence for any other tuple being in Big2. The rninirnal fixpoint conrputecl by stratified fixpoint evaluation is consistent \vith this intuition. However, there is another rninhnal fixpoint: We can place every part in Big2 and rnake Srna1l2 be ernpty. While this assignrllent of tuples to relations seeIns unintuitive, it is nonetheless a rninimal fixpoint.

rrhe requirernent that prograrns be stratified gives IIS a natural order for evaluating rules. When the rules are evaluated in this order, the result is a unique fixpoint that is one of the minirnal fixpoints of the prograrll. The fixpoint C0111puted by the stratified fixpoint evaluation usually corresponds well to our intuitive reading of a stratified prograrll, even if the program has rnore than one rllininlal fixpoint.

For nonstratified Da.talog progranls, it is harder to identify a natural model frorn arnong the alternative rninirnal rnodels, especially when we consider that the Ineaning of a prograrll must be clear even to users who lack expertise in Dlathelnatical logic. Although considerable research has been done on identifying natural rnodels for nonstratified prograrns, practical irnplernentations of Datalog have concentrated on stratified prograrns.

Relational Algebra and Stratified Datalog

Every relational algebra query can be written as a range-restricted, stratified Datalog progra.rn. (Of course, not all Datalog progranls can be expressed in relational algebra; for exarnple, the Cornponents prograrn.) 'We sketch the translation frorn algebra to stratified Datalog by writing a Datalog progra.rn for each of the basic algebra operations, in terrns of two eXC1mple tables R and S, each with two fields:

Selection:	Result(Y) :- Il(X,Y), X=c.
Projection:	Result(Y) :- H(X,Y).
Cross-product:	Result(X,Y,U,V) :- R(X,Y), S(lJ,V).
Set-difference:	Result(X,Y) :- R(X,Y), NOT S(U,V).
lJnion:	H.esult(X,Y) :- R(X,Y).
	Result(X,Y) :- S(X,Y).

We conclude ()ur discussion of stratification l>y noting that SQL:1999 requires prograrns to be stratified. rrhe stratified Big/Sruall prograrn is shown below in SQL:1999 notation, with a final additional selection on Big2:

SQL:1999 and Datalog Queries: A Datalog rule is **linear** recursive if the body contains at Illost one occurrence of any table that depends on the table in the head of the rule. A linear recursive program contains only linear recursive rules. All linear recursive Datalog progranls can be expressed using the recursive features of SQL:1999. IIowever, these features are not in Core SQL.

```
WITH
Big2(Part) AS
        (SELECT   A1.Part FROM Assernbly Al WHERE Qty > 2)
Srnall2(Part) AS
        ((SELECT- A2.Part FROM Assernbly A2)
        EXCEPT
        (SELECT   Bl.Part fron1 Big2 Bl))

SELECT * FROM Big2 B2
```

24.4 FROM DATALOG TO SQL

To support recursive queries in SQL,we lllust take into account the features of SQL that are not found in Datalog. Two central SQL features rnissing in Datalog are (1) SQL treats tables as *multisets* of tuples, rather than sets, and (2) SQL pennits grouping and aggregate operations.

The rnultiset selnantics of SQL queries can be preserved if we do not check for duplicates after applying rules. Every relation instance, including instances of the recursively defined tables, is a lllultiset. rrhe nurnber of occurrences of a tuple in a relation is equal to the nurnber of distinct inferences that generate this tuple.

The second point can be addressed by extending Data.logwith grouping and aggregation operations. Tlhis rnust be done\vith rnultiset sernantics in rnind, as we now illustrate. Consider the following prograrn:

```
NumPartsCPart, SUM((Qty))) :- AssemblyCPart, Subpart, Qty).
```

This prograrn is equivalent to the SQL query

```
SELECT    A.Part, SUM (A.Qty)
FROM      Assernbly A
GROUP BY  A.Part
```

The angular brackets (...) notation was introduced in the LDL deductive system, one of the pioneering deductive database prototypes developed at IVICC in the late 19808. We use it to dell0te *multiset generation*, or the creation of rnultiset-values. In principle, the rule definil1gNurnParts is evaluated by first creating the telnporary relation shO\vII in Figure 24.6. We create the ternporary relation by sorting on the *part* attribute (which appears on the left side of the rule, along with the (...) terrn) and collecting the I1ultiset of *qty* values for each *part* value. We then apply the SUM aggregate to each lllultiset-value in the second colu111n to obtain the ans\ver, \vhich is shown in Figure 24.7.

part	$\langle qty \rangle$
trike	{3,l}
frarne	{1,1}
wheel	{2,l}
tire	{1,1}

Figure 24.6 Temporary Relation

part	SUM ($\langle qty \rangle$)
trike	4
fran1e	2
wheel	3
tire	2

Figure 24.7 The Tuples in NumParts

The telnporary relation shown in Figure 24.6 need not be materialized to cornpute NurnParts; for exalllplc, SUM can be applied on-the-fly or Assenlbly can sirnply be sorted and aggregated as described in Section 14.6.

The use of grouping and aggregation, like negation, causes cOlnplicatiol1s when applied to a partially cOlnputed relation. rrhe difficulty is overcorne by adopting the sarne solution used for negation, stratification. Consider the following prograrn:[1]

```
TotParts(Part, Subpart, SUM«(Qty))) :- BOM(Part, Subpart, Qty).
BOM(Part, Subpart, Qty) :- Assembly(Part, Subpart, Qty).
BOM(Part, Subpart, Qty) :- Assembly(Part, Part2, Qty2),
                           BOM(Part2, Subpart, Qty3), Qty=Qty2*Qty3.
```

The idea is to count the l111mber of copies of Subpart for each Part. By aggregating over BOM rather than Assembly, we count subparts at any level in the hierarchy instead of just irnrnediate subparts. This prograrn is a version of a vvell-known problcrl1 called *Bill-of-Materials* and variants of it are probably the lnost widely used recursive queries in practice.

'rhe irnportant point to note in this exarnple is that we Inust wait until the relation BC)]VI has been cornpletely evaluated before we apply the rrotParts rule. Otherwise, \ve obta.in incornplete counts. T'his situation is analogous to theprobler11 we faced 'with negation; we have to evaluate the negated relation

[1]The reader should write this in SQL:1999 syntax, as a simple exercise.

SQL:1999 Cycle **Detection:** Safe Datalog queries that do not use arith-111etie operations have finite answers and the fixpoint evaluation is guaranteed to halt. Unfortunately, recursive SQL queries may have infinite answer sets and query evaluation may not halt. l'here are two independent reasons for this: (1) the use of aritlnnetie operations to generate data values that are not stored in input tables of a query, and (2) rTIultiset scrnantics for rule applications; intuitively, problems arise from *cycles* in the data. (To see this, consider the Cornponents prograrn on the Assenlbly instance shown in Figure 24.1 plus the tuple ⟨*tube, wheel,* 1).) SQL:1999 provides special constructs to check for such cycles.

cornpletely before applying a rule that involves the use of NOT. If a prograrn is stratified with respect to uses of ⟨...⟩ as well as NOT, stratified fixpoillt evaluation gives us 111eaningful results.

There are two further aspects to this exarnple. First, we rnust understand the cardinality of each tuple in BOlVI, based on the rnultiset sernantics for rule application. Second, we rnust understand the cardinality of the multiset of *Qty* values for each ⟨*Part, Subpart)* group in TotParts.

part	*subpart*	*qty*
trike	frarue	1
trike	seat	1
fra.rne	seat	1
frame	pedal	2
seat	cover	1

Figure 24.8 Another Instance of Assembly

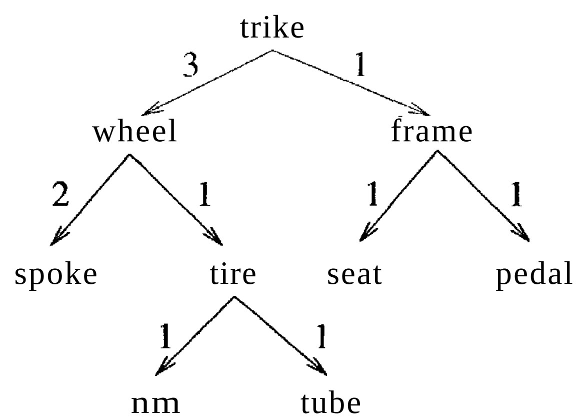


Figure 24.9 Assembly Instance Seen a,s a Graph

We illustrate these two points using the instance of Assernbly shown in Figures 24.8 and 24.9. f\pplying the first BOM rule, we add (one copy of) every tuple in Assernbly to BOM. Applying the second BOIVI rule, we add the follo\ving four tuples to BOM: ⟨*trike, seat,* 1), ⟨*trike, pedal, 2*⟩, ⟨*trike, cover,* 1), and ⟨*frame, cover,* 1). ()bserve that the tuple ⟨*trike, seat,* 1) was already in *BOlVf* because it was generated by applying the first rule; therefore, rnultiset sernantics for rule application gives us two copies of this tuple. Applying the second BC)IVI rule on the new tuples, we generate the tuple ⟨*trike, cover, 1*⟩ (using the tuple ⟨*frame, cover,* 1) for BaNI in t.he body of the rule): this is our second copy of the tuple. i\pplying the second rule again on this tuple does not generate any tuples, and the cOInputation of the BOM relation is now cOlnplete. The BaM instance at this stage is sho\vn in Figure 24.10.

part	*subpart*	*qty*
trike	frarne	1
trike	seat	1
frame	seat	1
fraIne	pedal	2
seat	cover	1
trike	seat	1
trike	pedal	2
trike	cover	1
frame	cover	1
trike	cover	1

Figure 24.10 Instance of *BON!* Table

part	*subpart*	*qty*
trike	fraIlle	{1}
trike	seat	{1,1}
trike	cover	{1,1}
trike	pedal	{2}
fram.e	seat	{1}
frame	pedal	{2}
seat	cover	{I}
frame	cover	{I}

Figure 24.11 Temporary Relation

Multiset grouping on this instance yields the temporary relation instance shown in Figure 24.11. (This step is only conceptual; the aggregation can be done on the fly without materializing this terllporary relation.) Applying SUM to the rllultisets in the third column of this temporary relation gives us the instance for TotParts.

24.5 EVALUATING RECURSIVE QUERIES

rrhe evaluation of recursive queries has been widely studied. While all the problems of evaluating nonrecursive queries continue to be present, the newly introduced fixpoint operation creates additional difficulties. A straightforward approach to evaluating recursive queries is to cornpute the fixpoint by repeatedly applying the rules as illustrated in Section 24.1.1. One application of all the prograrn rules is caIled an iteration; we perfonn as rnany iterations as necessary to reach the least fixpoint. This approach has two rnain disadvantages:

- Repeated Inferences: As Figures 24.3 and 24.4 illustrate, inferences are repeated across iterations. That is, the sarne tuple is inferred repeatedly in *the same way*, using the same rule and the same tuples for tables in the body of the rule.
- Unnecessary Inferences: Suppose we want to find the components of only a *wheel*. Cornputing the entire Cornponents table is wasteful and does not take advantage of inforrnation in the query.

In this section, we discuss how each of these difficulties can be overcorne. We consider only Datalog prograIns without negation.

24.5.1 Fixpoint Evaluation without Repeated Inferences

COlnputing the fixpoint by repeatedly applying all rules is called Naive fixpoint evaluation. Naive evaluation is guaranteed to cornpute the least fixpoint, but every application of a rule repeats all inferences lllade by earlier applications of this rule. We illustrate this point using the following rule:

Components(Part, Subpart) :- Assembly(Part, Part2, Qty),
Components(Part2, Subpart).

When this rule is applied for the first time, after applying the first rule defining Components, the Components table contains the projection of Assembly on the first two fields. Using these Components tuples in the body of the rule, we generate the tuples shown in Figure 24.3. For example, the tuple *(wheel, rim)* is generated through the following inference:

Components(wheel, rim) :- Assembly(wheel, tire, 1),
Components(tire, rim).

When this rule is applied a second time, the Components table contains the tuples shown in Figure 24.3 in addition to the tuples that it contained before the first application. Using the Components tuples shown in Figure 24.3 leads to new inferences; for example,

Components(trike, rim) :- Assembly(trike, wheel, 3),
Components(wheel, rim).

However, every inference carried out in the first application of this rule is also repeated in the second application of the rule, since all the Assernbly and Cornponents tuples used in the first rule application are considered again. For exarnple, the inference of *(wheel, rim⟩* shown above is repeated in the second application of this rule.

The solution to this repetition of inferences consists of rernelnbering which inferences were carried out in earlier rule applications and not carrying theln out again. We can 'remember' previously executed inferences efficiently by sirnply keeping track of which COlnponents tuples were generated for the first time in the rnost recent application of the recursive rule. Suppose we keep track by introducing a new relation called *delta_Components* and storing just the newly generated Cornponents tuples in it. Now, we can use only the tuples in *delta_Components* in the next application of the recursive rule; any inference using other COIuponents tuples should have been carried out in earlier rule applications.

This refincrllcnt of fixpoint evaluation is called Seminaive fixpoint evaluation.Let us trace Serninaive fixpoint evaluation on our exarllple program. The first application of the recursive rule produces the Cornponents tuples shown in Figure 24.3, just like Naive fixpoint evaluation, and these tuples are placed in *delta_Components*. In the second application, however, only *delta_Components* tuples are considered, which rneans that only the following inferences are carried out in the second application of the recursive rule:

```
Components(trike, rim)  :- Assembly(trike, wheel, 3),
                           delta_Components(wheel, rim).
Components(trike, tube) :-Assembly(trike, wheel, 3),
                           delta_Components(wheel, tube).
```

Next, the bookkeeping relation *delta_Cornponents* is updated to contain just these two Cornponents tuples. In the third application of the recursive rule, only these two *delta_Cornponents* tuples are considered and therefore no additional inferences can be nlade. The fixpoint of Cornponents has been reached.

To irnplernent Serninaive fixpoint evaluation for general Datalog prograrns, we apply all the recursive rules in a prograrll together in an iteration. Iterative application of all recursive rules is repeated until no new tuples are generated in SOHle iteration. To surnrnarize how Serninaive fixpoint evaluation is carried out, there are two irnportant differences with respect to Naive fixpoint evaluation:

- We rnaintain a *delta* version of every recursive predicate to keep track of the tuples generated for this predicate in the Inost recent iteration; for example, *delta_Cornponents* for COHlponents. The *delta* versions are updated at the end of each iteration.

- The original prograrn rules are re\vritten to ensure that every inference uses at least one *delta* tuple; that is, one tuple that\vas not kno\vn before the previous iteration. This property guarantees that the inference could not have been carried out in earlier iterations.

We do lI0t discuss details of Serninaive fixpoint evaluation (such as the algoritlun for rewriting progranl rules to ensure the use of a *delta* tuple in each inference).

24.5.2 Pushing Selections to Avoid Irrelevant Inferences

Consider a nonrecursive view definition. If we want only those tuples in the view that satisfy an additional selection condition, the selection can be added to the plan as a final operation, and the relational algebra transforlnations for conunuting selections with other relational operators allow us to 'push' the selection ahead of rnore expensive operations such as cross-products (;l,nd joins. In effect, we restrict the cornputation by utilizing selections in the query specification. The problerIl is rnore cOlnplicated for recursively defined queries.

We use the following progranl as an exarnple in this section:

```
8ameLevel(81 , 82)      Assembly(P1, 81, Q1),
                        Assembly(Pl, 82, Q2),
8ameLevel(81 , 82)      Assembly(Pl, 81, Qi),
                        8ameLevel(Pl, P2), Assembly(P2, 82, Q2).
```

Consider the tree representation of Assernbly tuples illustrated in Figure 24.2. 1"here is a tuple $\langle S1, S2\rangle$ in SarneLevel if there is a path froln 81 to 82 that goes up a certain nUlnber of edges in the tree and then CaInes down the saIne nurnber of edges.

Suppose we want to find all SalneLevel tuples with the first field equal to *spoke*. Since SalneLevel tuples can be used to COlupute other SarneLevel tuples, we cannot just cornpute those tuples with *spoke* in the first field. For exa.rnple, the tuple $\langle$*wheel*, *frarne*) in SarneLevel allows us to infer a SarneLevel tuple with *spoke* in the first field:

```
8ameLevel(spoke, seat) '-  Assembly(wheel, spoke, 2),
                           8ameLevel(wheel, frame),
                           Assembly(frame, seat, i),
```

Intuitively, we have to conlpute all SarneLevel tllpleswhose first field conta,ins a value on the path froln *spoke* to the root in Figure 24.2. Each such tuple has the potential to contribute to answers for the given query. On the other hand, cornputing the entire SameLevel table is wasteful; for exarnple, the SalneLevel tuple $\langle tire, seat\rangle$ cannot be used to infer any answer to the given query (or, indeed, to infer any tuple that can in turn be used to infer an answer tuple). We define a new table, \vhich we call l\1agic_SaIneLevel, such that each t11ple in this table identifies a value m for which we have to cornpute all SarneLevel tuples with m in the first colulun to answer the given query:

```
Magic_SameLevel(Pi) :- Magic_SameLevel(81), Assembly(P1, 81, Ql).
Magic_SameLevel(spoke) '-
```

Consider the tuples in lVlagic_SanleLevel. Obviously we have *(spoke)*. Using this lVlagic_SalneLevel tuple and the Assernbly tuple ⟨*wheel*, *spoke*, 2⟩, we can infer that the tuple ⟨*wheel*⟩ is in Magic_SameLevel. lJsing this tuple aJld the Assernbly tuple ⟨*trike*, '*wheel*, 3⟩, we can infer that the tuple ⟨*trike*⟩ is in Nlagic_SarneLevel. Thus, Magic_SameLevel contains each node that is on the path frorn *spoke* to the root in Figure 24.2. The Magic_SarneLcvel table can be llsed as a filter to restrict the computation:

```
5ameLevel(51 , 52)  :- Magic_5ameLevel(51) ,
            Assembly(P1, 51, Q1), Assembly(P2, 52, Q2).
5ameLevel(51 , 52)  :- Magic_SameLevel(S1), Assembly(Pl, 51, Ql),
            5ameLevel(Pl, P2), Assembly(P2, 52, Q2).
```

These rules together with the rules defining rvlagic_SarneLevel give us a progranl for cornputing all SanleLevel tuples with *spoke* in the first column. Notice that the new progranl depends on the query constant *spoke* only in the second rule defining lVlagic_SameLevel. Therefore, the program for cornputing all SameLevel tuples with *seat* in the first column, for instance, is identical except that the second Magic_SarneLevel rule is

```
Magic_5ameLevel(seat)  :- .
```

The nurnber of inferences rnade llsing the Magic program can be far fewer than the nurnber of inferences nlade using the original progranl, depending on just how rnuch the selection in the query restricts the cornputation.

24.5.3 The Magic Sets Algorithm

We illustrated the intuition behind the **Magic Sets** algorithrn on the SarneLevel prograrn, which contains just one output relation and one recursive rule.

The intuition behind the rewriting is that the rows in the Magic tables correspond to the subqueries whose answers are relevant to the original query. By evaluating the rewritten prograrn instead of the original prograrn, we can restrict cornputation by intuitively pushing the selection condition in the query into the recursion.

rIhe algorithrn, however, can be applied to any Datalog prograrn. The input to the algorithrn consists of the prograrn and a **query pattern**, which is a relation we want to query plus the fields for which a query will provide constants. The output of the algorithrn is a rewritten prograrn.

The Magic Sets program rewriting algorithrn can be surnrnarized as follows:

1. Generate the Adorned Prograln: In this step, the progranl is re\vritten to l1lake the pattern of queries and subqueries explicit.
2. Add Magic Filters: IVlodify each rule in the Adorned Prograrn by adding a IVlagic condition to the body that acts as a filter on the set of tuples generated by this rule.
3. Define the Magic Tables: We create new rules to define the Magic tables. Intuitively, froIll each occurrence of a table R in the body of an Adorned PrograIu rule, we obtain a rule defining the table Magic_R.

When a query is posed, we add the corresponding Magic tuple to the rewritten prograrl1 and evaluate the least fixpoint of the progrарIl (using Serninaive evaluation).

We rernark that the Magic Sets algorithrll has turned out to be quite effective for cornputing correlated nested SQL queries, even if there is no recursion, and is used for this purpose in rnany cornrnercial DBlVISs, even systenls that do not currently support recursive queries.

We now describe the three steps in the Magic Sets algorithrIl using the SarneLevel program as a running exalllple.

Adorned Program

We consider the query pattern $SameLevel^{bf}$. Thus, given a value c, we want to cornpute all rows in *SarneLevel* in which c appears in the first eolurnn. We generate the Adorned Prograrn P^{ad} frorn the given prograrn P by repeatedly generating adorned versions of rules in [J for every reachable query pattern, with the given query pattern as the only reachable pattern to begin with; additional reachable patterns are identified during the course of generating the A,dorned Prograrll as described next.

Consider a rule in? whose head contains the sarne table as sorne reachable pattern. rrhe adorned version of the rule depends on the order in \vhichwe consider the predicates in the body of the rule. To sirnplify our discussion, we assurne that this is always left-to-right. First, we replace the head of the rule with the rnatching query pattern. After this step, the recursive SarneLevel rule looks like this:

$SarneLevel^{bf}$ (S1, 82) :- Assembly(P1, 81, Q1),
8ameLevel(P1, P2), Assembly(P2, 82, Q2).

Next, we proceed left-to-right in the l)ody of the rule until we encounter the first recursive predicate. All cohullns that contain a constant or a variable that appears to the left are marked *b* (for *bound*) and the rest are rnar,ked *f* (for *free*) in the query pattern for this occurrence of the predicate. We add this pattern to the set of reachal)le patterns and Inodify the rule accordingly:

SameLevel^{bf}(S1, S2) :- Assembly(Pl, 81, Ql),
  SameLevel^{bf}(Pi, P2), Assembly CP2, 82, Q2).

If there are additional occurrences of recursive predicates in the body of the recursive rule, we continue (adding the query patterns to the reachable set and rllodifying the rule). (()f course, in linear recursive progralns, there is at illOSt one occurrence of a recursive predicate in a rule body.)

\Ve repeat this until we have generated the adorned version of every rule in *P* for every reachable query pattern that contains the same table as the head of the rule. The result is the Adorned Program *pad*, which, in our example, is

SameLeveZ^{bf}C81, 82) :- AssemblyCP1, 81, Ql),
  AssemblyCP1, 82, Q2).
SameLevel^{bf}(81, 82) :- AssemblyCP1, 81, Q1),
  SameLeveZ^{bf}CP1, P2), Assembly CP2, 82, Q2).

In our exarnple, there is only one reachable query pattern. In general, there can be several.[2]

Adding Magic **Filters**

Every rule in the Adorned Prograrn is rllodified by adding a 'nlagic filter' predicate to obtain the rewritten prograrn:

Sarne-Level^{bf}(81, 82) :- *Magic_SameLevel*^{bf}(81) ,
  Assembly(Pl, 81, Q1), Assembly(P2, 82, Q2).
SarneLeveZ^{bf}(S1, S2) :- *Magic_SameLevel*^{bf}CS1),
  Assembly(Pl, 81, Ql), *8arneLevel*^{bf}*(P1,* P2),
  Assembly(P2, 82, Q2).

The filter predicate is a copy of the head of the rule, 'with 'IVlagic' as a prefix for the table nàme and the variables in colllrnns corresponding to *free* deleted, as illustrated in these two rules.

[2] As an example, consider a variant of the SameLevel program in which the variables PI and $P2$ are interchanged in the body of the recursive rule (Exercise 24.5)

Defining Magic Filter Tables

Consider the Adorned Prograrl1 after every rule has been rnodified as described. FrorH each occurrence O of a recursive predicate in the body of a rule in this rIlodified prograrl1, we generate a rule that defines a Magic predicate. T'he algorithrl1 for generating this rule is as fo11o\:vs: (1) Delete everything to the right of occurrence () in the body of the rllodified rule. (2) Add the prefix 'Magic' and delete the free colulnns of (). (3) Move O, with these changes, into the head of the rule.

From the recursive rule in our example, after steps (1) and (2) we get:

$Sam,eLevel^{bf}$(S1, 82) :- $Magic\_SarneLevel^{bf}$(S1),
Assembly(P1, S1, Q1), $Magic\_SameLevel^{bf}$(P1) .

After step (3), we get:

$Magic\_SameLevel^{bf}$(P1) :- $Magic\_SameLevel^{bf}$(S1),
Assembly(Pl, S1, Q1).

The query itself generates a row in the corresponding Magic table, for exarnple, $Magic\_SarneLevel^{bf}$(seat).

24.6 REVIEW QUESTIONS

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

- Describe *Datalog* prograrIlS. lJse an exarnple Datalog prograrn to explain why it is not possible to write recursive rules in SQL-92. (Section 24.1)
- Define the terrns *rnodel* and *least model.* What can you say about least rnodels for Datalog prograrns? Why is this approach to defining the meaning of a Datalog prograrll called *declarative*? (Section 24.2.1)
- Define the tenns *.fi:rpoint* and *least ji:Epoint.* \iVhat can you say about least fixpoints for IJatalog prograrlls? vVhy is this approach to defining the rneaning of a Datalog prograrIl said to be *operational?* (Section 24.2.2)
- \Vhat is a *safe* prograrn? Why is this property irIlportant? What is *range-restriction* and how does it ensure safety'? (Section 24.2.3)
- \Vhat is the connection between least lnodels and least fixpoints for Datalog prograrIls? (Section 24.2.4)

- Explain why prograIlls with negation rnay not have a least model or least fixpoint. Extend the definition of *Tange-Testriction* to prograrns with negation. (Section 24.3)
- What is a *stratified* prograIn? How does stratification address the probleln of identifying a desired fixpoint? Show how every relational algebra query can be \vritten as a stratified Datalog prograrIl. (Section 24.3.1)
- Two important aspects of SQL, *multiset table8* and *aggr'egation 'with grouping,* are rnissing in Datalog. How can we extend Datalog to support these features? Discuss the interaction of these two new features and the need for stratification of aggregation. (Section 24.4)
- Define the terms *infeTence* and *iteration.* What are the two main challenges in efficient evaluation of recursive Datalog programs? (Section 24.5)
- Describe *Sem,inaive fixpoint evaluation* and explain how it avoids repeated inferences. (Section 24.5.1)
- Describe the *Magic Sets* program transformation and explain how it avoids unnecessary inferences. (Sections 24.5.2 and 24.5.3)

EXERCISES

Exercise 24.1 Consider the Flights relation:

> Flights(*fino:* integer, *from:* string, *to:* string, *distance:* integer,
> *departs:* time, *arrives:* time)

Write the following queries in Datalog and SQL:1999 syntax:

1. Find the *fino* of all flights that depart from Madison.
2. Find the *flno* of all flights that leave Chicago after Flight 101 arrives in Chicago and no later than 1 hour after.
3. Find the *fino* of all flights that do not depart from Madison.
4. Find aJI cities reachable frOlll Madison through a series of one or 1I10re connecting flights.
5. Find all cities reachable from IVladison through a chain of one or rnore connecting flights, with no 1I1Ore than 1 hour spent on any connection. (That is, every connecting flight must depart within an hour of the arrival of the previous flight in the chain.)
6. Find the shortest tilne to fly frOl11 Madison to Madras, using a chain of one or 1nore connecting flights.
7. Find the *Jlno* of all flights that do not depart [1'0111 Madison or a city that is reacha.ble from Madison through a chain of flights.

Exercise 24.2 Consider the definition of Cornponents in Section 24.1.1. Suppose that the second rule is replaced by

```
Components(Part, Subpart) :-Components(Part, Part2),
                            Components(Part2, Subpart).
```

1. If the rnodified program is evaluated on the ASS€lnbly relation in Figure 24.1, how Inany iterations does Naive fixpoint evaluation take and what COlllponents facts are generated in each iteration?
2. Extend the given instance of Assel1lbly so that Naive fixpoint iteration takes two rnore iterations.
3. Write this program in SQL:1999 syntax, using the WITH clause.
4. Write a progranl in Datalog syntax to find the part with the Inost distinct subparts; if several parts have the saIne Inaxinlllm number of subparts, your query should return all these parts.
5. How would your answer to the previous part be changed if you also wanted to list the number of subparts for the part with the Inost distinct subparts?
6. Rewrite your answers to the previous two parts in SQL:1999 syntax.
7. Suppose that you want to find the part with the rnost subparts, taking into account the quantity of each subpart used in a part, how would you rllodify the COlnponents program? *(Hint:* To write such a query you reason about the nuruber of inferences of a fact. For this, you have to rely on SQL's nlaintaining as many copies of each fact as the nurnber of inferences of that fact and take into account the properties of Seulinaive evaluation.)

Exercise 24.3 Consider the definition of Components in Exercise 24.2. Suppose that the recursive rule is rewritten as follows for Seminaive fixpoint evaluation:

```
Components(Part, Subpart) :- deLta_Components(Part, Part2, Qty),
                             deLta_Components(Part2, Subpart).
```

1. At the end of an iteration, what steps Illust be taken to update *delta_Cornponents* to contain just the new tuples generated in this iteration? Can you suggest an index on Cornponents that Inight help to make this faster?
2. Even if the *delta* relation is correctly updated, fixpoint evaluation using the preceding rule does not always produce all answers. Show an instance of Assembly that illustrates the probleru.
3. Can you suggest a way to rewrite the recursive rule in tenns of *delta_*Components so that Scrninaive fixpoint evaluation always produces all answers and no inferences are repeated across iterations?
4. Show how your version of the rewritten prograrn perfonns on the example instaJICe of Assernbly that you used to illustrate the problern with the gi"ven rewriting of the recursive rule.

Exercise 24.4 Consider the definition of SarneLevel In Section 24.5.2 and the Assernbly instance shown in Figure 24.1.

1. Rewrite the recursive rule for Seminaive fixpoint evaluation and show ho\v Serninaive evaluation proceeds.
2. Consider the rules defining the relation Magic, with *spoke* as the query constant. For Sernil1aive evaluation of the 'Magic' version of the SarneLevel prognllu, all tuples in Magic are cornputed first. Show how 8erninaive evaluation of the Magic relation proceeds.

3. After the Magic relation is cmnputed, it can be treated as a fixed database relation, just like Assembly, in the Senlinaive fixpoint evaluation of the rules defining SameLevel in the 'Magic' version of the prograrn. Rewrite the recursive rule for Selninaive evaluation and show how Scrninaive evaluation of these rules proceeds.

Exercise 24.5 Consider the definition of SanleLevel in Section 24.5.2 and a query in which the first argulnent is bound. Suppose that the recursive rule is rc\vritten as follows, leading to rnultiple binding patterns in the adorned program:

```
8ameLevel(81, S2) :- Assembly(Pl, 81, Ql),
          Assembly(P1, 82, Q2).
8ameLevel(81, S2) :- Assembly(Pl, S1, Ql),
          SameLevel(P2, P1), Assembly(P2, S2, Q2).
```

1. Show the adorned progranl.
2. Show the Magic program.
3. Show the Magic program after applying Seminaive rewriting.
4. Construct an example instance of Assenlbly such that the evaluating the optirnized prograrn generates less than 1% of the facts generated by evaluating the original prograrn (and finally selecting the query result).

Exercise 24.6 Again, consider the definition of SameLevel in Section 24.5.2 and a query in which the first argurnent is bound. Suppose that the recursive rule is rewritten as follows:

```
SameLevel(Sl, 82) :- Assembly(Pl, 81, Ql),
          Assembly(Pl, S2, Q2).
SameLevel(Sl, 82) :- Assembly(P1, S1, Ql),
          SameLevel(P1, Rl), SameLevel(Rl, P2), Assembly(P2, S2, Q2).
```

1. Show the adorned program.
2. Show the l\!Iagic prograln.
3. Show the Magic prograul after applying Serninaive rewriting.
4. Construct an exarnple instance of Asselnbly such that the evaluating the optimized progranl generates less than 1% of the facts generated by evaluating the original progranl (and finally selecting the query result).

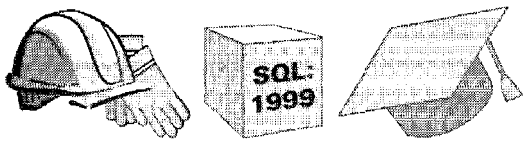
BIBLIOGRAPHIC NOTES

The use of logic as a query language is discussed in several papers [296, 537], "which arose out of influential workshops. Good textbook discussions of deductive databases can be found in [747, 3, 143, 794, 503]. [614] is a recent survey article that provides an overview and covers the rnajor prototypes in the area, including LI)L [177], Glue-Nail! [214, 549] EKS-Vl [758], Aditi [615], Coral [612], LOLA [804], and XSB [644].

The fixpoint sernantics of logic programs (and deductive databases as a special case) is presented in [751], which also shows equivalence of the fixpoint selnantics to a *least-model* semantics. The use of stratification to give a natural sernantics to prograrns with negation was developed independently in [37, 154, 559,752].

Efficient evaluation of deductive database queries has been widely studied, and [58J is a survey and cOlnparison of several early techniques; [611] is a more recent survey_ Serninaive fixpoint evaluation was independently proposed several tiInes; a good treatment appears in [54]. r1'he Magic Sets technique is proposed in [57] and generalized to cover all deductive database queries without negation in [77]. The Alexander rnethod [631] was independently developed and is equivalent to a variant of Magic Sets called *Supplementary Magic Sets* in [77]. [553] shows how Magic Sets offers significant perfonnance benefits even for nonrecursive SQL queries. [673] describes a version of Magic Sets designed for SQL queries with correlation, and its irnplernentation in the Starbufst systern (which led to its iInplenlentation in IBNI's DB2 DBNIS). [670] discusses how Magic Sets can be incorporated into a Systenl R style cost-based optimization framework. The Magic Sets technique is extended to prograIIls with stratified negation in [53, 76]_ [121] cOlnpares Magic Sets with top-do\vn evaluation strategies derived froIn Prolog.

[642] develops a prograrn rewriting technique related to Magic Sets called *Magic Counting*. Other related methods that are not based on progranl rewriting but rather on fun-tirne control strategies for evaluation include [226, 429, 756, 757]. The ideas in 1.226] have been developed further to design an *abstract rnachine* for logic prograll1 evaluation using tabling in [609, 727]; this is the basis for the XSB systell1 [644].



25

DATA WAREHOUSING AND DECISION SUPPORT

- Why are traditional DBIvISs inadequate for decision support?
- What is the multidimensional data nlOdel and what kinds of analysis does it facilitate?
- What SQL:1999 features support rnultidiInensional queries?
- How does SQL:1999 support analysis of sequences and trends?
- How are DBMSs being optimized to deliver early answers for interactive analysis?
- What kinds of index and file organizations do OLAP systerlls require?
- What is data warehousing and why is it irnportant for decision support?
- Why have rnaterialized views becorne iInportant?
- How can we efficiently Inaintain rnaterialized views?

➽ Key concepts: OLAP, rnultirnensional rnodel, dimensions, measures; roll-up, drill-clown, pivoting, cross-tabulation, CUBE; WINDOW queries, frames, order; top N queries, online aggregation; bitmap indexes, join indexes; data warehouses, extract, refresh, purge; rnaterialized views, incremental rnaintenancc, rnaintaining warehouse views

Notlling is lnore difficult, and therefore more precious, than to be able to decide.

. NCtI)oleon Bonaparte

Database luanagclncnt systerIls are widely used by organizations for rnaintaining data that docurnents their everyday operations. In applications that update such *operational data*, transactions typically rnake sma11 changes (for exalnple, adding a reservation or depositing a check) and a large nU111ber of transactions H1USt be reliably and efficiently processed. Such **online transaction processing (OLTP)** applications have driven the gruwth of the DBMS industry in the past three decades and will doubtless continue to be irnportant. DB1VISs have traditionally been optirnized extensively to perforn1 well in such applications.

H,ecently, ho\vever, organizations have increasingly crnphasized applications in which current and historical data is coruprehensively analyzed and explored, identifying useful trends and creating sununaries of the data, in order to support high-level decision rnaking. Such applications are referred to as **decision support**. Mainstream relational DBMS vendors have recognized the irnportance of this rnarket segment and are adding features to their products to support it. In particular, SQL has been extended with new constructs and novel indexing and query optirl1ization techniques are being added to support cornplex queries.

The use of views has gained rapidly in popularity because of their utility in applications involving cornplex data analysis. While queries on views can be answered by evaluating the view definition when the query is subrnitted, precornputing the view definition can rnake queries run Inuch faster. Carrying the r11otivation for preconlputed views one step further, organizations can consolidate inforrnation from several databases into a *data warehouse* by copying tables fror11 rnany sources into one location or rnaterializing a view defined over tables fr01n several sources. Data '\varehousing has becorne widespread, and Il1any specialized products are no\v available to create and rnanage warehouses of data frorH 1l1ultiple databases.

We begin this chapter with an overview of decision support in Section 25.1. We introduce the rnultirnensional rnodel of data in Section 25.2 and consider database design issues in 25.2.1. We discuss the rich class of queries that it naturally supports in Section 25.;3. We discuss how new SQL:1999 constructs allow us tel express rnultidilnensional queries in 25.3.1. In Section 25.4, we discuss SQL:1999 extensions that support queries over relations as ordered collections. We consider how to optimize for fast generation of initial answers in Sectioll 25.5. The rnany query language extensions required in the ()LA.P envirolllnentprornpted the developrnent of llC\V irnplcrncntation techniques; we discuss these in Section 25.6. In Section 25.7, \ve examine the issues involved in creating and rnaintaining a data \varehouse. FraIn a technical standpoint, a key issue is how to maintain warehouse inforrnation (replicated tables or views) ·when the llnderl.ying source infonnation changes. After covering the important role played byvic\vs in OLAP and warehousing irl Section 25.8, we consider maintenance of rnaterialized views in Sections 25.9 and 25.10.

25.1 INTRODUCTION TO DECISION SUPPORT

()rganizational decisioll rnaking requires a cOlnprehensive view of all aspects of an enterprise, so many organizations created consolidated **data warehouses** that contain data drawn frcHn several databases ll1atntained by different business units together with historical and summary inforInation.

The trend toward data warehousing is c0I11plelnented by an increased ernphasis on powerful analysis tools. Many characteristics of decision support queries make traditional SQL systenls inadequate:

- 1'he WHERE clause often contains rnany AND and OR conditions. As we saw in Section 14.2.3, OR conditions, in particular, are poorly handled in rnany relational DBMSs.
- Applications require extensive use of statistical functions, such as standard deviation, that are not supported in SQL-92. Therefore, SQL queries rnust frequently be ernbedded in a host language program.
- Many queries involve conditions over time or require aggregating over time periods. SQL-92 provides poor support for such time-series analysis.
- Users often need to pose several related queries. Since there is no convenient way to express these cOlnnlonly occurring families of queries, users have to write thern as a collection of independent queries, \vhich can be tedious. Further, the DBMS has no way to recognize and exploit optimization opportunities arising froln executing nlany related queries together.

Three broad classes of analysis tools are available. First, SOlne systerIls support a class of stylized queries that typically involve group-by and aggregation operators and provide excellent support for cOlnplex boolean conditions, statistical functions, and features for tilne-series analysis. Applications dominated by such queries are called **online analytic processing (OLAP)**. 'These systerns support a querying style in which the data is best thought of as a rnultidilnensional array and are influenced by end-user tools, such as spreadsheets, in addition to database query languages.

Second, sorne DBMSs support traditional SQL-style queries but are designed to also support OLAP queries efficiently. Such systenls can be regarded as relational DBMSs optirnized for decision support applications. Many vendors of relational DBIVISs are currently enhancing their products in this direction and, over tilne, the distinction between specialized OLAP systerns and relational DBIVISs enhanced to support ()LAP queries is likely to dirninish.

The third class of analysis tools is rllotivated by the desire to find interesting or unexpected trends and patterns in large data sets rather than the conlplex

SQL:1999 and OLAP: In this chapter, we discuss a nUInber of features introduced in SQL:1999 to support OLAP. In order not to delay publication of the SQL:1999 standard, these features \vere actually added to the standard through an *amendment* called SQL/OLAP.

query characteristics just listed. In exploratory data analysis, although an analyst can recognize an :interesting pattern' when shown such a pattern, it is very difficult to fannulate a query that captures the essence of an interesting pattern. For exalnple, an analyst looking at credit-card usage histories Illay want to detect unusual activity indicating Inisuse of a lost or stolen card. A catalog lllerchant lnay want to look at custolner records to identify pro111ising custoiners for a new proillotion; this identification would depend on inccnIle level, buying patterns, delllonstrated interest areas, and so all. The alllount of data in Inany applications is too large to perrnit rnanual analysis or even traditional statistical analysis, and the goal of data mining is to support exploratory analysis over very large data sets. We discuss data rnining further in Chapter 26.

Clearly, evaluating OLAP or data rnining queries over globally distributed data is likely to be excruciatingly slow. Further, for such cOlnplex analysis, often statistical in nature, it is not essential that the IllOSt current version of the data be used. The natural solution is to create a centralized repository of all the data; that is, a data warehouse. Thus, the availability of a warehouse facilitates the application of ()LAP and data rnining tools and, conversely, the desire to apply such analysis tools is a strong 1Ilotivation for building a data warehouse.

25.2 OLAP: MULTIDIMENSIONAL DATA MODEL

aLAI' applications are dOlninated by ad hoc, cOlnplex queries. In SQL terllls, these are queries that involve group-by and aggregation operators. The natural way to think about typical ()LAP queries, ho\vever, is in tenns of a rnultidilnensinnal data rllodel. In this section, we present the rnultidirnensional data Illodel and corupare it with a relational representation of data. In subsequent sections, we describe ()LAP queries in. terrns of the rllultidirnensional data rnodel and consider some new irnplernentation techniques designed to support such queries.

In the rnultidirnensional data rnodel, the focus is on a collection of nurneric measures. Each Ineasure depends on a set of dirnensions. We use a running exarnple based on sales data. The measure attribute in our exarnple is *sales*. The dirnensions are Product, Location, and Tirne. Given a product, a location;

and a tinle, we have at I110st one associated sales value. If we identify a product by a unique identifier *pid* and, sirnilarly, identify location by *locid* and time by *timeid*, we can think of sales infornation as being arranged in a three-dirnensional array Sales. This array is shown in Figure 25.1; for clarity, we show only the values for a single *locid* value, *locid-* 1, which can be thought of as a slice orthogonal to the *lacid* axis.

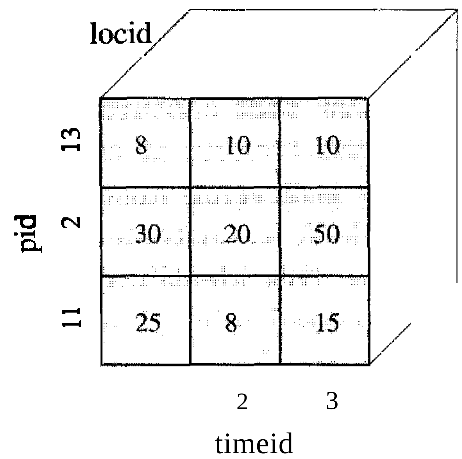


Figure 25.1 Sales: A Multidimensional Dataset

'This view of data as a multiclhnensional array is readily generalized to rnore than three dirnensions. In OLAP applications, the bulk of the data can be represented in such a rnultidiInensional array. Indeed, some OLAP systerns actually store data in a rnultidiInensional array (of course, irnplenlented without the usual prograrnrning language asslunption that the entire array fits in rnelnory). OLAP systerns that use arrays to store rnultidirnensiona.l datasets are called nlultidimensional OLAP (MOLAP) systcrl1s.

The data in a 11lultidirnensional array can also be represented as a relation, as illustrated in Figure 25.2, which shows the same data as in Figure 25.1, with additional rows corresponding to the 'slice' *locid=* 2. Tlhis relation, which relates the dirnensions to the rneasure of interest, is called the fact table.

Now let us tun1 to dirnensions.Each dirnension can have a set of associated attributes. For exarnple, the Location dilTlension is identified by the *loc'id* attribute, which we used to identify a location in the Sales table. We aSSUlnc that it also has attributes *country*, *state*, and *city*. We further assurne that the Product dirnension has attributes *pname*, *category*, and *price* in additi(Jn to the identifier *pid*. 'rhe *category* of a product indicates its general nature; for exarnple, a product *pant* could have category value *apparel*. We assurne that the Time dirnension has attributes *date*, *week*, *month*, *quarter*, *year*, and *holiday_flag* in addition to the identifier *timeid*.

pid	*timeid*	*locid*	*sales*
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Sales

locid	*city*	*state*	*country*
1	Madison	WI	USA
2	Fresno	CA	USA
5	Chennai	TN	India

Locations

pid	*pname*	*category*	*price*
11	Lee Jeans	Apparel	25
12	Zord	Toys	18
13	Biro Pen	Stationery	2

Products

Figure 25.2 Locations, Products, and Sales Represented as Helations

For each dinlension, the set of associated values can be structured as a hierarchy. For exarnple, cities belong to states, and states belong to countries. Dates belong to weeks and rIlonths, both 'weeks and 1110nths are contained in quarters, and quarters are contained in years. (Note that a week could span two rnonths; therefore, weeks are not contained in rnonths.) SCHne of the attributes of a diruension describe the position of a dirnensioll valuevvith respect to this underlying hierarchy of dirnensioll values. The hierarchies for the Product, Location, and Time hierarchies in our exarnple are sho\vn at the attribute level in Figure 25.3.

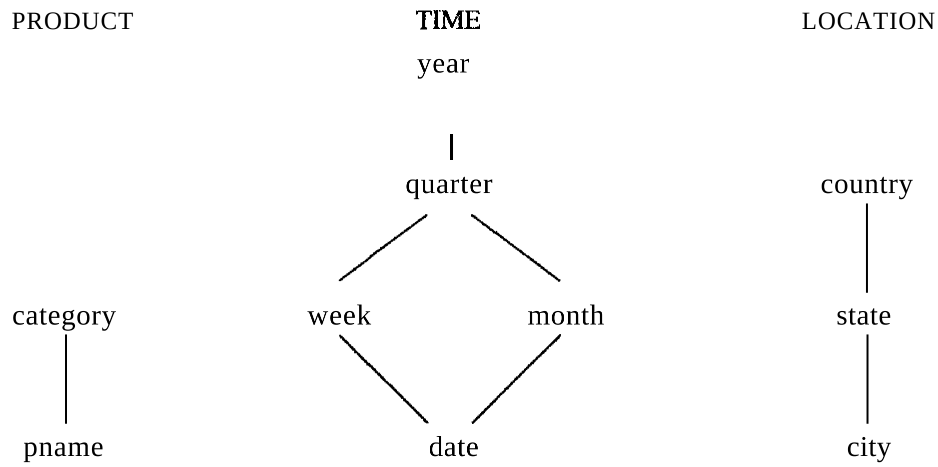


Figure 25.3 Dimension Hierarchies

Infonnation about dirnensions can also be represented as a collection of relations:

> Locations(*locid:* integer, *city:* string, *state:* string, *country:* string)
> *Products(pid:* integer, *pname:* string, *category:* string, *price:* real)
> Times(*timeid:* integer, *date:* string, *week:* integer, *rnonth:* integer,
> *quarter:* integer, *year:* integer, *holiday_fiag:* boolean)

These relations arc luuch srnalIer than the fact table in a typical 0 I..lAP application; they are called the **dimension** tables. OLAP systcrl1s that store all inforrnation, including fact tables, as relations are called relational OLAP **(ROLAP)** systcrns.

The Tinlcs table illustrates the attention paid to the T'irne dirnension in typical OLAP applications. SQL's date and tirnestaulp data types are not adequate; to support slunrnarizations that reflect business operations, infonnation such as fiscal quarters, holiday status, and so on is rnaintained for each tirne value.

25.2.1 Multidimensional Database Design

Figure 25.4 shows the tables in our running sales example. It suggests a star, centered at the fact table Sales; such a cornbination of a fact table and dirncnsion tables is called a **star schema**. This schelna pattern is very COIIUIlon in databases designed for 0 LAP. The bulk of the data is typically in the fact table, which has no redundancy; it is usually in BCNF. In fact, to Ininimize the size of the fact table, dirnension identifiers (such as *p'id* and *timeid*) are systcrn-generated identifiers.

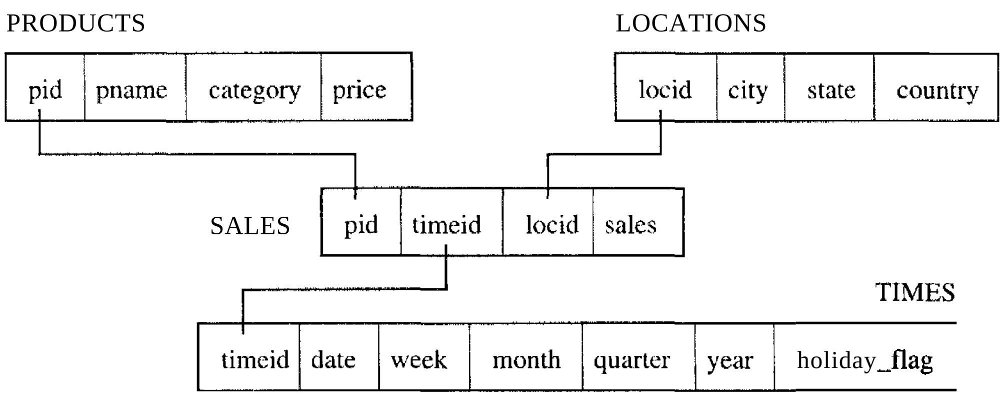


Figure 25.4 An Example of a Star Schema

Inforrnation about dinlension values is rnaintained in the dirnension tables. Di111ension tables are usually not nonnalized. The rationale is that the dimension tables in a database used for OL,AP are static and update, insertion, and deletion anoillalies are not irnportant. Further, because the size of the database is dorninated by the fact table, the space saved by norrnalizing dilnension tables is negligible. Therefore, mini111izing the cornputation tilllC for cOlllbining facts in the fact table with dirnension inforrnation is the rnain design criterion, which suggests that we avoid breaking a dirnension table into srnaller tables (which rnight lead to additional joins).

Snlall response tirnes for interactive querying are irnportant in OLAP, and rnost systerns support the Hlaterialization of SUrl1Inary tables (typically generated through queries using grouping). Ad hoc queries posed by users are answered using the original ta,bles along with precornputed surnrnaries. A very irnportant design issue is which sunnnary tables should be rnaterialized to achieve the best use of available rnerllory and answer cOHnI1only asked ad hoc queries with interactive response tirnes. In current OLAP systerns, deciding which surnnlary tables to rnaterialize rnay well be the Inost irnportant design decision.

Finally, new storage structures and indexing techniques have been developed to support ()LAP and they present the database designer \'lith additional physical design choices. We cover BOIHe of these hnplclnentatiol1 techniques in Section 25.6.

25.3 MULTIDIMENSIONAL AGGREGATION QUERIES

Now that we have seen the rnulticliInensiol1alluoclel of data, let us consider how such data can be queried and rnanipulatecl. The operations supported by this Inodel are strongly influenced by end user tools such as spreadsheets. The goal is to give end users who are not SQL experts an intuitive and po\verful interface for cornnlon business-oriented analysis tasks. Users are expected to pose ad hoc queries directly, without relying on database application prograrrnners.

In this section, we asslllne that the user is working with a multidirnensional dataset and that each operation returns either a different presentation or a sunllnary; the underlying dataset is always available for the user to 1nanipulate, regardless of the level of detail at which it is currently viewed. In Section 25.3.1, we discuss how SQL:1999 provides constructs to express the kinds of queries presented in this section over tabular, relational data.

A very C01111non operation is aggregating a rneasure over one or 1nore dimensions. The following queries are typical:

- Find the total sales.
- Find total sales for each city.
- Find total sales for each state.

'These queries can be expressed as SQL queries over the fact and dirnension tables. When we aggregate a measure Oll one or rnore di1nensions, the aggregated measure depends on fewer diInensioIls than the original measure. For example, when we cornpute the total sales by city, the aggregated rneasure is *total sales* and it depends only on the Location di1nension,whereas the original *sales* rneasure depended on the Locatioll,Tirne, a,nd Product dirnensions.

Another use of aggregation is to SU1Ilrnarize at different levels of a dirnension hierarchy. If we are given total sales per city, we can aggregate 011 the Location dinlension to obtain sales per state. This operation is called roll-up in the OLAI' literature. The inverse of roll-up is drill-down: Given total sales by state, we can ask for a 1Ilore detailed presentation by drilling down on Location. We can ask for sales by city 01 just sales by city f01 a selected state (with sales presented on a per-state basis for the rernaining states, as before). We can also drill dowll on a diluension other than Location. For example, we can ask for total sales for each product for each state, drilling down OII the Prodnet diInension.

Another C0111ll10n operation is pivoting. Consider a, tabular presentation of the Sales table. If we pivot it on the Location and Titne dirnensions, we obtain a table of total sales for each location for each tillle value. This infoI"luation can be presented as a two-dimensional chart in which the axes are labeled 'with location and time values; the entries in the chart correspond to the total sales for that location and time. Therefore, values that appear in colurnns of the original presentation becoIne labels of axes in the result presentation. The result of pivoting, called a **cross-tabulation**, is illustrated in Figure 25.5. Observe that in spreadsheet style, in addition to the total sales by year and state (taken together), we also have additional sunlillaries of sales by year and sales by state.

	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	399

Figure 25.5 Cross-Tabulation of Sales by Year and State

Pivoting can also be used to change the dirnensions of the cross-tabulation; froIn a presentation of sales by year and state, we can obtain a presentation of sales by product and year.

Clearly, the OLAP frarnework rnakes it convenient to pose a broad class of queries. It also gives catchy naInes to sorne farniliar operations: **Slicing** a dataset arnonnts to an equality selection on one or rIlore dirnensions, possibly also with SC)lne dirnensions projected out. **Dicing** a dataset arl10unts to a range selection. These terrllS corne frcnl1 visuaJizing the effect of these operations on a cube or cross-tabulated representation of the data.

A Note on Statistical Databases

Many ()LAP concepts are present in earlier work on **statistical databases** (SDBs), which are database systerl1s designed to support statistical applications, although this connection has not been sufficiently recognized because of differences in application dornains and tern.linology. The rnultidirnensional data rllodel, 'with the notions of a rneasure associated with dirnensions and classification hierarchies for dirncIlsion vahles, is also used in SDBs. OLAP operations such as roll-up and drill-down have counterparts in SDBs. Indeed, sorne irnplcrnentation techniques developed for OLAP are also applied to SDBs.

Nonetheless, some differences arise frorn the different dOlnains OLAP and SDBs were developed to support. For exarnple, SnBs are used in socioeconornic applications, where classification hierarchies and privacy issues are very ilnportant. This is reflected in the greater cornplexity of classification hierarchies in SDBs, along with issues such as potential breaches of privacy. (The privacy issue concerns whether a user with access to sUllunarized data can reconstruct the original, unsununarized data.) In contrast, OLAP has been ailned at business applications with large volulnes of data and efficient handling of very large datasets has received lnore attention than in the SDB literature.

25.3.1 ROLLUP and **CUBE** in SQL:1999

In this section, we discuss how lnany of the query capabilities of the rnultidi-111ensionalrllloclel are supported in SQL:1999. Typically, a single OLAP operation leads to several closely related SQL queries with aggregation and grouping. For exarnple, consider the cross-tabulation shown in Figure 25.5, which was obtained by pivoting the Sales table. To obtain the saIne inforrnation, we would issue the following queries:

```
SELECT    rr.year, 1.state, SUM (S.sales)
FROM      Sales S, T'irnes T, Locations L
WHERE     S.tirneid=T.tiIneid AND S.locid=L.locid
GROUP BY  T.year, 1.state
```

This query generates the entries in the body of the chart (outlined by the dark lines). The surllluary cohunn on the right is generated by the query:

```
SELECT    T.year, SUM (S.saJes)
FROM      Sales S, Times T
WHERE     S.timeid= T.tuncld
GROUP BY  T.year
```

The sunnnary ro\v at the bottorl1 is generated l)y the query:

```
SELECT    L.state, SUM (S.sales)
FROM      Sales S, Locations L
WHERE     S.locid=L.locicl
GROUP BY  L.state
```

The cumulative suITI in the bottonl-right corner of the chart is produced by the query:

```
SELECT   SUM (S.sales)
FROM     Sales S, Locations L
WHERE    S.loc:id=L.locid
```

The exarnple cross-tabulation can be thought of as roll-up on the entire dataset (i.e., treating everything as one big group), on the Location dirnension, on the rrirne dirnensioIl, and on the Location and Tinle dinlensions together. Each roll-up corresponds to a single SQL query with grouping. In general, given a measure with k associated dirnensions, we can roll up on any subset of these k diInensions; so we have a total of 2^k such SQL queries.

Through high-level operations such as pivoting, users can generate lTlany of these 2^k SQL queries. R,ecognizing the cornrnonalities between these queries enables r110re efficient, coordinated COlTlputation of the set of queries.

SQL:1999 extends the GROUP BY construct to provide better support for roll-up and cross-tabulation queries. The GROUP BY clause with the CUBE keyword is equivalent to a collection of GROUP BY statenlents, with one GROUP BY statement for each subset of the k dirnensions.

Consider the following query:

```
SELECT    rr.year, L.state, SUM (S.sales)
FROM      Sales S, Tirnes T, Locations L
WHERE     S.tirneid=T'.tirneid AND S.1ocid=L.locid
GROUP BY  CUBE (T.year, L.state)
```

The result of this query, shown in Figure 25.6, is just a tabular representation of the cross-tabulation in Figure 25.5.

SQL:1999 also provides variants of GROUP BY that enable cornputatioll of subsets of the cross-tabulation cornputed using GROUP BY CUBE. For example, we can replace the grouping clause in the previous query with

```
GROUP BY ROLLUP (T.year, L.state)
```

In contrast to GROUP BY CUBE, vile aggregate by an pairs of year and state values and by each year, and cornpute an overall sIIHI for the entire dataset (the last row in Figure 25.6), but we do not aggregate for each state value. The result is identical to that shown inF'igure 25.6, except that the rows with *null* in the *T. year* cOhUIIII and non-*null* values in the *L.state* colurnn are not cornputed.

```
CUBE pid, locid, tirneid BY SUM Sales
```

T.year	L.state	SUM(S.sales)
1995	WI	63
1995	CA	81
1995	*null*	144
1996	WI	38
1996	CA	107
1996	*null*	145
1997	WI	75
1997	CA	35
1997	*null*	110
null	WI	176
null	CA	223
null	*null*	399

Figure 25.6 The Result of GROUP BY CUBE on Sales

rrhis query rolls up the table Sales on all eight subsets of the set {pid, locid, tirneid} (including the empty subset). It is equivalent to eight queries of the fonn

```
SELECT     SUM (S.sales)
FROM       Sales S
GROUP BY   grouping-list
```

The queries differ only in the *grouping-list,* which is sorne subset of the set {pid, locid, tirneid}. We can think of these eight queries as being arranged in a lattice, as shown in Figure 25.7. The result tuples at a node can be aggregated further to cornpute the result for any child of the node. This relationship between the queries arising in a CUBE can be exploited for efficient evaluation.

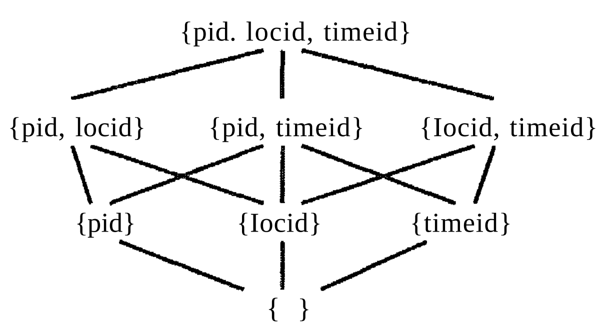


Figure 25.7 The Lattice of GROUP BY Queries ill a CUBE Query

25.4 WINDOW QUERIES IN SQL:1999

The time dirnension is very important in decision support and queries involving trend analysis have traditionally been difficult to express in SQL. To address this, SQL:1999 introduced a fundamental extension called a query window. Examples of queries that can be written using this extension, but are either difficult or iInpossible to write in SQL without it, include

1. Find total sales by rnonth.
2. Find total sales by rnonth for each city.
3. Find the percentage change in the total monthly sales for each product.
4. Find the top five products ranked by total sales.
5. Find the trailing *n* day moving average of sales. (For each day, we must compute the average daily sales over the preceding *n* days.)
6. Find the top five products ranked by cumulative sales, for every month over the past year.
7. Rank all products by total sales over the past year, and, for each product, print the difference in total sales relative to the product ranked behind it.

The first two queries can be expressed as SQL queries using GROUP BY over the fact and dinlension tables. The next two queries can be expressed too, but are quite complicated in SQL-92. The fifth query cannot be expressed in SQL-92 if *n* is to be a pararneter of the query. The last query cannot be expressed in SQL-92.

In this section, we discuss the features of SQL:1999 that allow us to express all these queries and, obviously, a rich class of sirnilar queries.

The rnain extension is the WINDOW clause, which intuitively identifies an ordered 'window' of rows 'around' each tuple in a table. This allows us to apply a rich collection of aggregate functions to the windovv of a row and extend the row with the results. For exarnple, we can associate the average sales over the past 3 days with every Sales tuple (each of which records 1 day's sales). This gives us a 3-day Illoving average of sales.

\Vhile there is sorne sirnilarity to the GROUP BY and CUBE clauses, there are important differences as well. For exarnple, like the WINDOW operator, GROUP BY allows us to create partitions of rows and apply aggregate functions such as SUM to the rows in a partition. However, unlike WINDOW, there is a single output row per pa.rtition, rather than one output row for each row, and each partition is an unordered collection of rows.

We now illustrate the window concept through an exalnple:

```
SELECT L.state, T.month, AVG (S.sales) OVER W AS Inovavg
FROM   Sales S, Tinles T, Locations L
WHERE  S.tirneid=T.tirIleid AND S.1ocid=L.locid
WINDOW W AS (PARTITION BY L.state
       ORDER BY 'f.lnonth
       RANGE BETWEEN INTERVAL '1' MONTH PRECEDING
       AND INTERVAL '1' MONTH FOLLOWING)
```

The FROM and WHERE clauses are processed as usual to (conceptually) generate an interrnediate table, which we refer to as Ternp. Windows are created over the TeHIp relation.

There are three steps in defining a window. First, we define *part'it'ions* of the table, using the PARTITION BY clause. In the exarnple, partitions are based on the *L.8tate* colurnn. Partitions are similar to groups created with GROUP BY, but there is a very important difference in how they are processed. To understand the difference, observe that the SELECT clause contains a column, *T. month,* which is not used to define the partitions; different rows in a given partition could have different values in this colulun. Such a colurnn cannot appear in the SELECT clause in conjunction with grouping, but it is allowed for partitions. 'The reason is that there is one answer row for *each* row in a partition of Ternp, rather than just one answer row per partition. The window around a given row is used to COlnpute the aggregate functions in the corresponding answer row.

The second step in defining a \vindow is to specify the *ordering* of rows within a partition. We do this using the ORDER BY clause; in the exarnple, the rows within each partition are ordered by *T. month.*

The third step in window definition is to *frame* windo\vs; that is, to establish the boundaries of the window associated with each row in terrns of the ordering of rows within partitions. In the exalnple, the window for a row includes the row itself plus all rows whose rnonth value is within a Inonth before or after; therefore, a row whose *Tnonth* value is Jllne 2002 has a window containing all rows with *Tnonth* equal to May, June, or July 2002.

1'he answer row corresponding to a given row is constructed by first identifying its \vindo\v. Then, for each answer colurun defined using a window aggregate function, we cornpute the aggregate llsing the ro\vs in the window.

In our example, each row of Temp is essentially a ro\v of Sales, tagged with extra details (about the location and tirne dirnensions). There is one partition for ea.ch state and every ro\v of Ternp belongs to exactly one partition. Consider a ro\v for a store in \Visconsin. The row states the sales for a given product, in that store, at a certain tirHe. The window for this row includes all rows that describe sales in \Visconsin within the previous or next Inonth and *movavg* is the average of sales (over all products) in \Visconsin \vithin this period.

We note that the ordering of rows within a partition for the purposes of window definition does not extend to the table of answer ro\vs. The ordering of answer rows is nondeterlninistic, unless, of course, we fetch therIl through (1, cursor and use ORDER BY to order the cursor's output.

25.4.1 **Framing** a Window

There are two distinct ways to fra1118 a window in SQL:1999. The exarnple query illustrated the RANGE construct, which defines a window based on the values in SOllle cohulln *(rnonth* in our exarnple). The ordering colu111n has to be a nU111eric type, a datetillle type, or an interval type since these are the only types for which addition and subtraction are defined.

The second approach is based on using the ordering directly and specifying how Illany rows before and after the given row are in its window. Thus, we could say

```
SELECT  L.state, T.rnonth, AVG (S.sales) OVER W AS Inovavg
FROM    Sales S, Times T, Locations L
WHERE   S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
        ORDER BY T.IIlonth
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
```

If there is exactly one row in Tenlp for each IIlonth, this is equivalent to the previous query. IIo\vever, if a given lnonth has no rows or lnultiple rows, the t\VO queries produce different results. In this case, the result of the second query is hard to understand because the \vindc)\vs for different rows do not align in a natural way.

The second approach is appropriate if, in terms of our example, there is exactly one 1'o\v per lllonth. C·eneralizing frOIrI this, it is also appropriate if there is exactly one row for every value in the sequence of ordering COhlI1UI values. ·UnJike the first approach, 'where the ordering has to be specified over a single (rullneric, datetime, or interval type) colurnn, the ordering can be based on a cornposite key.

We can also define \vindows that include all rows that are before a given row (UNBOUNDED PRECEDING) or all ro\vs after a given row (UNBOUNDED FOLLOWING) 'within the row's partition.

25.4.2 New Aggregate Functions

While the standard aggregate functions that apply to rnultisets of values (e.g., SUM, AVG) can be used in conjunction \vith Willdo\ving, there is a lleed for a new class of functions that operate on a *list* of values.

The RANK function returns the position of a row within its partition. If a partition has 15 rows, the first row (according to the ordering of rows in the window definition over this partition) has rank 1 and the last row has rank 15. The rank of intermediate rows depends on whether there are multiple (or no) rows for a given value of the order.ing colurnn.

Consider our running example. If the first row in the Wisconsin partition has the lllonth January 2002, and the second and third rows both have the rnonth February 2002, then their ranks are 1, 2, and 2, respectively. If the next row has rllonth March 2002 its rank is 4.

In contrast, the DENSE_.RANK function generates ranks without gaps. In our exalnple, the four rows are given ranks 1, 2, 2, and 3. The only change is in the fourth row, whose rank is now 3 rather than 4.

The PERCENT_RANK function gives a lneasure of the relative position of a row within a partition. It is defined as (RANK-1) divided by the Innnber of rows in the partition. CUME_DIST is sirnilar but based on actual position within the ordered partition rather than rank.

25.5 FINDING ANSWERS QUICKLY

A recent trend, fueled in part by the popularity of the Internet, is an ernphasis 011 queries for which a user wants only the first few, or the 'best' few, answers quickly. When users pose queries to a search engine such as AltaVista, they rarely look beyond the first or second page of results. If they do not find what they are looking for, they refine their query and resubrnit it. The same phen()lneuon occurs in decision support applications and scnne DBl\;1S products (e.g., DB2) already support extended SQL con.structs to specify sueh queries. A related trend is that, for cornplex queries, users would like to see an approximate answer quickly and then have it be continually refined, rather than \vait until the exact answer is available. We now discuss these two trends l)riefly.

25.5.1 Top N Queries

An analyst often wants to identify the top-selling handful of products, for example. We can sort by sales for each product and return answers in this order. If we have a Inillion products and the analyst is interested only in the top 10, this straightforward evaluation strategy is clearly \vasteful. It is desirable for users to be able to explicitly indicate how rnany answers they want, rnaking it possible for the DBMS to optirnize execution. l-'he follo\ving exarnple query asks for the top 10 products ordered by sales in a given location and time:

```
SELECT      P.pid, P.pnarne, S.sales
FROM        Sales S, Products P
WHERE       S.pid=P.pid AND S.locid==l AND S.tiIneid=3
ORDER BY    S.sales DESC
OPTIMIZE    FOR 10 ROWS
```

The OPTIMIZE FOR N ROWS construct is not in SQL-92 (or even SQL:1999), but it is supported in IBM's DB2 product, and other products (e.g., Oracle 9i) have sirnilar constructs. In the absence of a cue such as OPTIMIZE FOR 10 ROWS, the DBMS computes sales for all products and returns thenl in descending order by sales. The application can close the result cursor (i.e., tenninate the query execution) after consulning 10 rows, but considerable effort has already been expended in cornputing sales for all products and sorting them.

Now let us consider how a DBMS can use the OPTIMIZE FOR cue to execute the query efficiently. The key is to sOlnehow cornpute sales only for products that are likely to be in the top 10 by sales. Suppose that we know the distribution of sales values because we rnaintain a histogran1 on the *sales* cohuun of the Sales relation. We can then choose a value of *sales*, say, c, such that only 10 products have a larger sales value. For those Sales tuples that meet this condition, we can apply the location and tirne conditions as well and sort the result..Evaluating the following query is equivalent to this approach:

```
SELECT      P.pid, P.pnarne, S.sales
FROM        Sales S, Products P
WHERE       S.pid=P.picl AND S.locid=1 AND S.timeid=3 AND S.sales > c
ORDER BY    S.sales DESC
```

This approach is, of course, ruuch faster than the alternative of cornputing all product sales and sorting thern, but there are SOIne in1portant problerns to resolve:

1. *flow do we choose the sales cutoff value* c? EIistograrns and other systeln statistics can be used for this m1rl)()SC, but this can be a tricky issue. For

one thing, the statistics rnaintained by a DBMS are only approximate. For another, even if we choose the cutoff to reflect the top 10 sales values accurately, other conditions in the query Inay elirninate SOHle of the selected tuples, leaving us with fewer than 10 tuples in the result.

2. *What 'if we have more than 10 tuples in the result?* Since the choice of the cutoff c is approxirnate, we could get 1nore than the desired nurnber of tuples in the result. rrhis is easily handled by returning just the top 10 to the user. We still save considerably with respect to the approach of cornputing sales for all products, thanks to the conservative pruning of irrelevant sales infonnation, using the cutoff c.

3. *What 'if we have fewer than 10 tuples in the result?* Even if we choose the sales cutoff c conservatively, we could still cOlnpute fe\ver than 10 result tuples. In this case, we can re-execute the query with a srnaller cutofF value c_2 or simply re-execute the original query with no cutoff.

The effectiveness of the approach depends on how well we can estirnate the cutoff and, in particular, on rninimizing the nurnber of tiules we obtain fewer than the desired nurnber of result tuples.

25.5.2 Online Aggregation

Consider the following query, which asks for the average sales arIlount by state:

```
SELECT    L.state, AVG (S.sales)
FROM      Sales S, Locations L
WHERE     S.locid=L.locid
GROUP BY  L.state
```

This can be an expensive query if Sales and Locations are large relations. We cannot achieve fast response tirnes with the traditional approach of cornputing the anwer in its entirety when the query is presented. One alternative, as we have seen, is to use precornputation. Another alternative is to cornpute the answer to the query when the query is presented l)ut return an approximate answer to the user as soon as possible. As the cornputation progresses, the answer quality is continually refined. This approach is called online aggregation. It is very attra,ctive for queries involving aggregation, beca,use efficient techniques for cornputing and refining approxirnate answers are available.

Online aggregation is illustrated in Figure 25.8: For each state--the grouping criterion for our exarnple query · the current value for average sales is displayed, together with a confidence interval The entry for Alaska tells us that the

STATUS	PRIORITIZE	State	.VG(sales)	[illegible]	Interval
		Alabama	5,232.5	97%	103.4
		Alaska	2,832.5	93%	132.2
		Arizona	6,432.5	98%	52.3
		Wyoming	4,243.5		

Figure 25.8 Online Aggregation

current estiInate of average per-store sales in Alaska is $2,832.50, and that this is within the range $2,700.30 to $2,964.70 with 93% probability. rrhe status bar in the first column indicates how close we are to arriving at an exact value for the average sales and the second cohllnn indicates 'whether calculating the average sales for this state is a priority. Estimating average sales for Alaska is not a priority, but estimating it for Arizona is a priority. As the figure indicates, the DBMS devotes Inore systern resources to estiInating the average sales for high-priority states; the estirnate for Arizona is Inucll tighter than that for Alaska and holds with a higher probability. Users can set the priority for a state by clicking on the Prioritize button at any tilne during the execution. This degree of interactivity, together with the continuous feedback provided by the visual display, rnakes online aggregation an attractive technique.

To irnplernent online aggregation, a DEl\!IS lIlust incorporate statistical techniques to provide confidence intervals for approxiInate answers and use non-blocking algorithms for the relational operators. An algorithnl is said to block if it does not produce output tuples until it has consurned all its input tuples. For exarnple, the sort-Illerge join algoritlun blocks because sorting requires all input tuples before detennining the first output tuple. Nested loops join and hash join are therefore preferable to sort-rnerge join for online aggregation. Sirnilarly, hash-based aggregation is better than sort-based aggregation.

25.6 IMPLEMENTATION TECHNIQUES FOR OLAP

In this section we survey 80r11e irnplernentatioll techniques rllotivated by the ()LAP envirornnent. rrhe goal is to provide a feel for how ()LAP systerIls differ fron1 1nore traditional SQL systerns; our discussion is far frorn cornprehensive.

Beyond B+ Trees: Complex queries have rnotivated the addition of powerful indexing techniques to DBMSs. In addition to B+ tree indexes, Oracle 9i supports bitlnap and join indexes and Inaintains these dynalni-cally as the indexed relations are updated. Oracle 9i also supports indexes on expressions over attribute values, such as 10 * *sal* + *bonus*. Microsoft SQL Server uses bitrnap indexes. Sybase IQ supports several kinds of bitrnap indexes, and rnay shortly add support for a linear hashing based index. Informix UDS supports R trees and Inforrnix XPS supports bitlIlap indexes.

The rIlostly-read environruent of OLAP systerns rnakes the CPU overhead of rnaintaining indexes negligible and the requireruent of interactive response tinles for queries over very large datasets rnakes the availability of suitable indexes very important. This combination of factors has led to the developrnent of new indexing techniques. We discuss several of these techniques. We then consider file organizations and other OLAP implenlentation issues briefly.

We note that the ernphasis on query processing and decision support appli-cations in OLAP systems is being cornplemented by a greater erllphasis on evaluating cOlnplex SQL queries in traditional SQL systerIls. Traditional SQL systerns are evolving to support OLAP-style queries more efficiently, supporting constructs (e.g., CUBE and window functions) and incorporating irnpleruentation techniques previously found only in specialized 0 LAP systems.

25.6.1 Bitmap Indexes

Consider a table that describes custorners:

Custoruers(<u>*custid:* integer,</u> *narne:* string, *gender':* boolean, *rating:* integer)

The *rating* value is an integer in the range 1. to 5, and only two values are recorded for *gender*. Cohllnns with few possible values are called sparse. We can exploit sparsity to construct a new kind of index that greatly speeds up queries 011 these cobulins.

The idea is to record values for sparse colurnns as a sequence of bits, one for each possible value. For exarnple, a, *gender* value is either 10 or en; a 1. in the first position denotes ruale, and 1. in the second position denotes fe1nale. Similarly, 10000 denotes the *rai'ing* value 1, and 00001 denotes the *rating* value 5.

If we consider the *gender* values for all rows in the Custorners table, we can treat this as a collection of two bit vectors, OIle of which has the associated value M(ale) and the other the associated value F(ernale). Each bit vector has one bit per row in the Custorners table, indicating whether the value in that row is the value associated with the bit vector. The collection of bit vectors for a COhUIIII is called a bitrnap index for that colurnn.

An exaInple instance of the Customers table, together with the bitlnap indexes for *gender* and *rating,* is shown in Figure 25.9.

M	*F*
1	0
1	0
0	1
1	0

custid	*name*	*gender*	*rating*
112	Joe	M	3
115	RaIn	M	5
119	Sue	F	5
112	Woo	M	4

1	*2*	*3*	*4*	*5*
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Figure 25.9 Bitmap Indexes on the Customers Relation

Bitmap indexes offer two important advantages over conventional hash and tree indexes. First, they allow the use of efficient bit operations to answer queries. For example, consider the query, "How Inany Inale custolllers have a rating of 5?" We can take the first bit vector for *gender* and do a bitwise AND with the fifth bit vector for *rating* to obtain a bit vector that has 1 for every male custoIner with rating 5. We can then count the number of Is in this bit vector to answer the query. Second, bitmap indexes can be much luore cOInpact than a traditional B+ tree index and are very amenable to the use of cornpression techniques.

Bit vectors correspond closely to the rid-lists used to represent data entries in Alternative (3) for a traditional B+ tree index (see Section 8.2). In fact, we can think of a bit vector for a given *age* value, say, as an alternative representation of the rid-list for that value.

This suggests away to combine bit vectors (and their advantages of bitwise processing) with B+ tree indexes: We can use Alternative (3) for data entries, using a bit vector representation of rid-lists. A caveat is that, if an rid-list is very slnall, the bit vector representation rnay be Illuch larger than a list of rid values, even if the bit vector is cornpressed. Further, the use of corupression leads to decornprcssion costs, offsetting sorne of the C0I11putational advantages of the bit vector representation.

A Inore flexible approach is to usc a standard list representation of the rid-list for S01ne key values (intuitively, those that contain few clernents) and a bit vector representation for other key values (those that contain rnany elenlents, and therefore lend themselves to a cOInpact bit vector representation).

This hybrid approach, which can easily be adapted to work with hash indexes as well as B+ tree indexes, has both advantages and disadvantages relative to a standard list of rids approach:

1. It can be applied even to cohllnns that are not sparse; that is, in which are Tnany possible values can appear. The index levels (or the hashing scheIue) allow us to quickly find the 'list' of rids, in a standard list or bit vector representation, for a given key value.
2. Overall, the index is Tnore cornpact because we can use a bit vector representation for long rid lists. We also have the benefits of fast bit vector processIng.
3. On the other hand, the bit vector representation of an rid list relies on a Inapping fron1 a position in the vector to an rid. (This is true of any bit vector representation, not just the hybrid approach.) If the set of rows is static, and we do not worry about inserts and deletes of rows, it is straightforward to ensure this by assigning contiguous rids for rows in a table. If inserts and deletes Inust be supported, additional steps are required. For exanlple, we can continue to assign rids contiguously on a per-table basis and sirnply keep track of which rids correspond to deleted rows. Bit vectors can now be longer than the current nUlnber of rows, and periodic reorganization is required to cOlllpact the 'holes' in the assignrnent of rids.

25.6.2 Join Indexes

Cornputing joins with sIllall response tirnes is extrernely hard for very large relations. One approach to this problern is to create an index designed to speed up specific join queries. Suppose that the Custorners table is to be joined with (1, table called Purchases (recording purchases made by custorners) on the *custid* field.vVe can create a collection of $\langle c, p\rangle$ pairs, where p is the rid of a Purchases record that joins with a Custc)lners recol'c! with *custid* c.

This idea can be generalized to support joins over ruore than two relations. We discuss the special case of a star scherna., in which the fact table is likely to be joined with several dirnension tables. Consider a join query that joins fact table F with dilnension tables D1 and D2 and includes selection conditions on cohunn C_1 of tal)le 1)1 and colurnn (:1$_2$ of table D2. We store a tuple $\langle r_1$, ('2, $r\rangle$ irl the join index if *r1* is the rid of a tuple in table 1)1 with value ('1 in cohunn C_1, '12 is the rid of a tuple in table D2 with value *c2* in colllrnn C_2, and r is the rid of a tllple in the fact table F, and these three tUl)les join with each other.

Complex Queries: The IBM DB2 optimizer recognizes star join queries and perfOfIns rid-based sernijoins (using BIoarn filters) to filter the fact table. Then fact table rows are rejoined to the dimension tables. Cornplex (rnnltitable) dirnension queries (called *snowflake queries*) are supported. DB2 also supports CUBE using smart algorithrns that rninhnize sorts. Microsoft SQL Server optiInizes star join queries extensively. It considers taking the cross-product of srnall dirnension tables before joining with the fact table, the use of join indexes, and rid-based semijoins. Oracle 9i also allows users to create diInensions to declare hierarchies and functional dependencies. It supports the CUBE operator and optirnizes star join queries by elinlinating joins when no colunlll of **a** dirnension table is part of the query result. DBMS products have also been developed specifically for decision support applications, such as Sybase IQ.

The drawback of a join index is that the nurnber of indexes can grow rapidly if several colurnns in each dirnension table are involved in selections and joins with the fact table. An alternative kind of join index avoids this problem. Consider our exarnple involving fact table F and dirnension tables Dl and D2. Let G_1 be a column of Dl on which a selection is expressed in some query that joins Dl with F. Conceptually, we now join F with Dl to extend the fields of F with the fields of Dl, and index F on the 'virtual field' $G1$: If a tuple of Dl with value CI in colurnn C_1 joins with a tuple of F with rid r, we add a tuple $(C1' r\rangle$ to the join index. We create one such join index for each colurnn of either Dl or D2 that involves a selection in SOHle join with F; C_1 is an exarnple of such a COIUllUl.

The price paid with respect to the previous version of join indexes is that join indexes created in this way have to be cornbined (rid intersection) to deal with the join queries of interest to us. This can be done efficiently if we rnake the ne\v indexes *bitrnap* indexes; the result is called a, bitrnapped join index. The idea works especially well if cohunns such as C_1 are sparse, and therefore well suited to bitrnap indexing.

25.6.3 File Organizations

Since rllFtny OLAP queries involve just a few colurnns of a large relation, vertical partitioning becornes attractive. However, storing a. relation colurnn-\vise can degrade perfoI"rnance for queries that involve several colurnns. An alternative in a rl1ostly-read envirollrnent is to store the relation row-wise, but also store each COIUH1II separately.