Unit 3: Emerging Database Management System Technologies
Part II
(Fundamentals of Database Systems, Third Edition by Ramez Elmasri and Shamkant B. Navathe)

23.1 Active Database Concepts

Rules that specify actions that are automatically triggered by certain events have been considered as important enhancements to a database system for quite some time. In fact, the concept of triggers—a technique for specifying certain types of active rules—has existed in early versions of the SQL specification for relational databases. Commercial relational DBMSs—such as Oracle, DB2, and SYBASE—have had various versions of triggers available. However, much research into what a general model for active databases should look like has been done since the early models of triggers were proposed. We will use the syntax of the Oracle commercial relational DBMS to illustrate these concepts with specific examples, since Oracle triggers are close to the way rules will be specified in the SQL3 standard.

23.1.1 Generalized Model for Active Databases and Oracle Triggers

The model that has been used for specifying active database rules is referred to as the **Event-Condition-Action**, or **ECA** model. A rule in the ECA model has three components:

1. The **event** (or events) that trigger the rule: These events are usually database update operations that are explicitly applied to the database. However, in the general model, they could also be temporal events (An example would be a temporal event specified as a periodic time, such as: Trigger this rule every day at 5:30 am.) or other kinds of external events.

2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Let us consider some examples to illustrate these concepts. The examples are based on a much simplified variation of the COMPANY database application from Figure 07.07, which is shown in Figure 23.01, with each employee having a name (NAME), social security number (SSN), salary (SALARY), department to which they are currently assigned (DNO, a foreign key to DEPARTMENT), and a direct supervisor (SUPERVISOR_SSN, a (recursive) foreign key to EMPLOYEE). For this example, we assume that null is allowed for DNO, indicating that an employee may be temporarily unassigned to any department. Each department has a name (DNAME), number (DNO), the total salary of all employees assigned to the department (TOTAL_SAL), and a manager (MANAGER_SSN, a foreign key to EMPLOYEE).

EMPLOYEE

Name	Ssn	Salary	Dno	Supervisor_ssn

DEPARTMENT

Dname	Dno	Total_sal	Manager_ssn

Fig: A simplified COMPANY database used for active rule samples.

Notice that the TOTAL_SAL attribute is really a derived attribute, whose value should be the sum of the salaries of all employees who are assigned to the particular department. Maintaining the correct value of such a derived attribute can be done via an active rule. We first have to determine the events that may cause a change in the value of TOTAL_SAL, which are as follows:

1. Inserting (one or more) new employee tuples.
2. Changing the salary of (one or more) existing employees.
3. Changing the assignment of existing employees from one department to another.
4. Deleting (one or more) employee tuples.

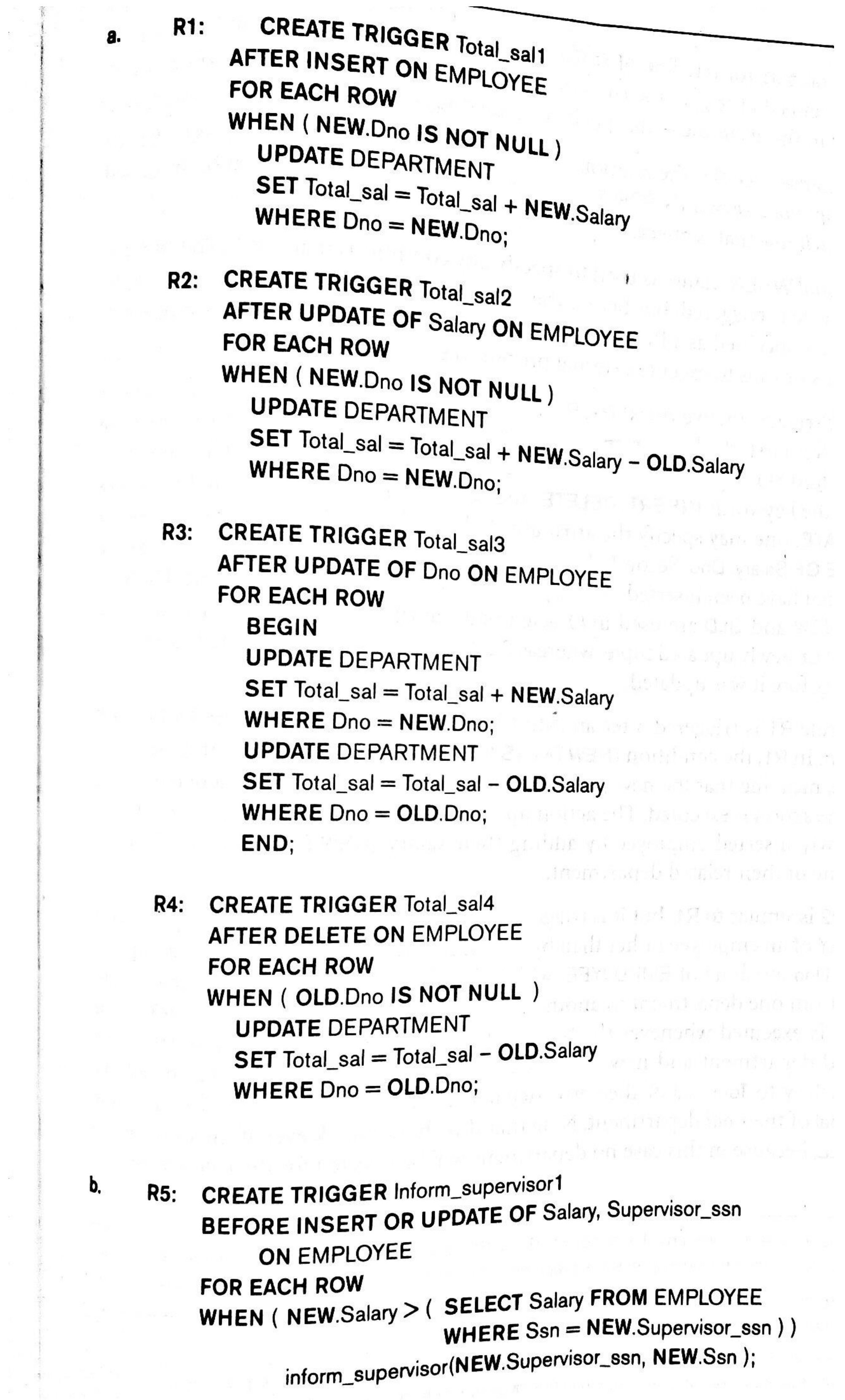
In the case of event 1, we only need to recompute TOTAL_SAL if the new employee is immediately assigned to a department—that is, if the value of the DNO attribute for the new employee tuple is not null (assuming null is allowed for DNO). Hence, this would be the **condition** to be checked. A similar condition could be checked for events 2 (and 4) to determine whether the employee whose salary is changed (or who is being deleted) is currently assigned to a department. For event 3, we will always execute an action to maintain the value of TOTAL_SAL correctly, so no condition is needed (the action is always executed).

The **action** for events 1, 2, and 4 is to automatically update the value of TOTAL_SAL for the employee's department to reflect the newly inserted, updated, or deleted employee's salary. In the case of event 3, a twofold action is needed; one to update the TOTAL_SAL of the employee's old department and the other to update the TOTAL_SAL of the employee's new department.

The four active rules R1, R2, R3, and R4—corresponding to the above situation—can be specified in the notation of the Oracle DBMS as shown in Figure 23.02(a). Let us consider rule R1 to illustrate the syntax of creating active rules in Oracle. The CREATE TRIGGER statement specifies a trigger (or active rule) name—TOTALSAL1 for R1. The AFTER-clause specifies that the rule will be triggered after the events that trigger the rule occur. The triggering events—an insert of a new employee in this example—are specified following the AFTER keyword (it is also possible to specify BEFORE instead of AFTER, which indicates that the rule is triggered before the triggering event is executed.).

The ON-clause specifies the relation on which the rule is specified—EMPLOYEE for R1. The optional keywords FOR EACH ROW specify that the rule will be triggered once for each row that is affected by the triggering event.

The optional WHEN-clause is used to specify any conditions that need to be checked after the rule is triggered but before the action is executed. Finally, the action(s) to be taken are specified as a PL/SQL block, which typically contains one or more SQL statements or calls to execute external procedures.



```
a.   R1:   CREATE TRIGGER Total_sal1
           AFTER INSERT ON EMPLOYEE
           FOR EACH ROW
           WHEN ( NEW.Dno IS NOT NULL )
              UPDATE DEPARTMENT
              SET Total_sal = Total_sal + NEW.Salary
              WHERE Dno = NEW.Dno;

     R2:   CREATE TRIGGER Total_sal2
           AFTER UPDATE OF Salary ON EMPLOYEE
           FOR EACH ROW
           WHEN ( NEW.Dno IS NOT NULL )
              UPDATE DEPARTMENT
              SET Total_sal = Total_sal + NEW.Salary – OLD.Salary
              WHERE Dno = NEW.Dno;

     R3:   CREATE TRIGGER Total_sal3
           AFTER UPDATE OF Dno ON EMPLOYEE
           FOR EACH ROW
              BEGIN
              UPDATE DEPARTMENT
              SET Total_sal = Total_sal + NEW.Salary
              WHERE Dno = NEW.Dno;
              UPDATE DEPARTMENT
              SET Total_sal = Total_sal – OLD.Salary
              WHERE Dno = OLD.Dno;
              END;

     R4:   CREATE TRIGGER Total_sal4
           AFTER DELETE ON EMPLOYEE
           FOR EACH ROW
           WHEN ( OLD.Dno IS NOT NULL )
              UPDATE DEPARTMENT
              SET Total_sal = Total_sal – OLD.Salary
              WHERE Dno = OLD.Dno;

b.   R5:   CREATE TRIGGER Inform_supervisor1
           BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn
                 ON EMPLOYEE
           FOR EACH ROW
           WHEN ( NEW.Salary > ( SELECT Salary FROM EMPLOYEE
                                 WHERE Ssn = NEW.Supervisor_ssn ) )
                    inform_supervisor(NEW.Supervisor_ssn, NEW.Ssn );
```

Fig: Specifying active rules as triggers in Oracle notation.

a) Triggers for automatically maintaining the consistency of Total_sal of DEPARTMENT.
b) Trigger for comparing an employee's salary with that of his or her superior

The four triggers (active rules) R1, R2, R3, and R4 illustrate a number of features of active rules. First, the basic events that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. These are specified by the keywords INSERT, DELETE, and UPDATE in Oracle notation. In the case of UPDATE one may specify the attributes to be updated—for example, by writing UPDATE OF SALARY, DNO. Second, the rule designer needs to have a way to refer to the tuples that have been inserted, deleted, or modified by the triggering event. The keywords NEW and OLD are used in Oracle notation; NEW is used to refer to a newly inserted or newly updated tuple, whereas OLD is used to refer to a deleted tuple or to a tuple before it was updated.
Thus rule R1 is triggered after an INSERT operation is applied to the EMPLOYEE relation. In R1, the condition (NEW.DNO IS NOT NULL) is checked, and if it evaluates to true, meaning that the newly inserted employee tuple is related to a department, then the action is executed. The action updates the DEPARTMENT tuple(s) related to the newly inserted employee by adding their salary (NEW.SALARY) to the TOTAL_SAL attribute of their related department.

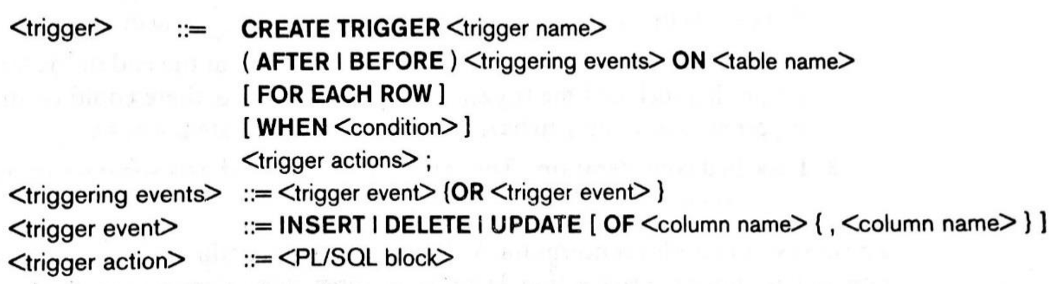
Rule R2 is similar to R1, but it is triggered by an UPDATE operation that updates the SALARY of an employee rather than by an INSERT. Rule R3 is triggered by an update to the DNO attribute of EMPLOYEE, which signifies changing an employee's assignment from one department to another. There is no condition to check in R3, so the action is executed whenever the triggering event occurs. The action updates both the old department and new department of the reassigned employees by adding their salary to TOTAL_SAL of their new department and subtracting their salary from TOTAL_SAL of their old department. Note that this should work even if the value of DNO was null, because in this case no department will be selected for the rule action.

It is important to note the effect of the optional FOR EACH ROW clause, which signifies that the rule is triggered separately for each tuple. This is known as a **row-level trigger**. If this clause was left out, the trigger would be known as a **statement-level trigger** and would be triggered once for each triggering statement. To see the difference, consider the following update operation, which gives a 10 percent raise to all employees assigned to department 5. This operation would be an event that triggers rule R2:

```
UPDATE   EMPLOYEE
  SET   SALARY = 1.1 * SALARY
  WHERE   DNO = 5;
```

Because the above statement could update multiple records, a rule using row-level semantics, such as R2 in Figure 23.02, would be triggered once for each row, whereas a rule using statement-level semantics is triggered only once. The Oracle system allows the user to choose which of the above two options is to be used for each rule. Including the optional FOR EACH ROW clause creates a row-level trigger, and leaving it out creates a statement-level trigger. Note that the keywords NEW and OLD can only be used with row-level triggers.

A syntax summary for specifying triggers in the Oracle system (main options only).



```
<trigger>           ::=  CREATE TRIGGER <trigger name>
                         ( AFTER | BEFORE ) <triggering events> ON <table name>
                         [ FOR EACH ROW ]
                         [ WHEN <condition> ]
                         <trigger actions> ;
<triggering events>  ::= <trigger event> {OR <trigger event> }
<trigger event>      ::= INSERT | DELETE | UPDATE [ OF <column name> { , <column name> } ]
<trigger action>     ::= <PL/SQL block>
```

23.1.2 Design and Implementation Issues for Active Databases

The previous section gave an overview of the main concepts for specifying active rules. In this section, we discuss some additional issues concerning how rules are designed and implemented. The first issue concerns activation, deactivation, and grouping of rules. In addition to creating rules, an active database system should allow users to activate, deactivate, and drop rules by referring to their rule names. A deactivated rule will not be triggered by the triggering event. This feature allows users to selectively deactivate rules for certain periods of time when they are not needed. The activate command will make the rule active again. The drop command deletes the rule from the system.

Another option is to group rules into named rule sets, so the whole set of rules could be activated, deactivated, or dropped. It is also useful to have a command that can trigger a rule or rule set via an explicit PROCESS RULES command issued by the user.

The second issue concerns whether the triggered action should be executed before, after, or concurrently with the triggering event. A related issue is whether the action being executed should be considered as a separate transaction or whether it should be part of the same transaction that triggered the rule. We will first try to categorize the various options. It is important to note that not all options may be available for a particular active database system. In fact, most commercial systems are limited to one or two of the options that we will now discuss.

Let us assume that the triggering event occurs as part of a transaction execution. We should first consider the various options for how the triggering event is related to the evaluation of the rule's condition. The rule condition evaluation is also known as rule consideration, since the action is to be executed only after considering whether the condition evaluates to true or false. There are three main possibilities for rule consideration:

1. **Immediate consideration**: The condition is evaluated as part of the same transaction as the triggering event, and is evaluated immediately. This case can be further categorized into three options:
   - Evaluate the condition before executing the triggering event.
   - Evaluate the condition after executing the triggering event.
   - Evaluate the condition instead of executing the triggering event.

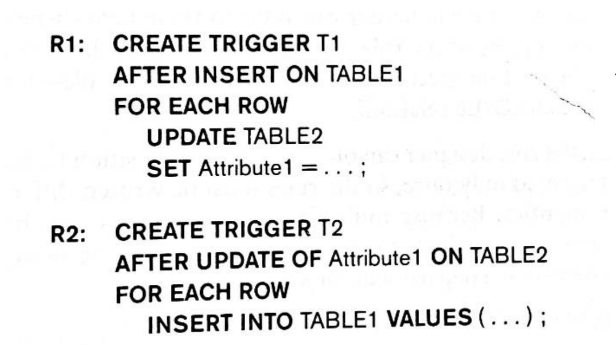
2. **Deferred consideration**: The condition is evaluated at the end of the transaction that included the triggering event. In this case, there could be many triggered rules waiting to have their conditions evaluated.

3. **Detached consideration**: The condition is evaluated as a separate transaction, spawned from the triggering transaction.

The next set of options concern the relationship between evaluating the rule condition and executing the rule action. Here, again, three options are possible: immediate, deferred, and detached execution. However, most active systems use the first option. That is, as soon as the condition is evaluated, if it returns true, the action is immediately executed.

One of the difficulties that may have limited the widespread use of active rules, in spite of their potential to simplify database and software development, is that there are no easy-to-use techniques for designing, writing, and verifying rules. For example, it is quite difficult to verify that a set of rules is consistent, meaning that two or more rules in the set do not contradict one another. It is also difficult to guarantee termination of a set of rules under all circumstances. To briefly illustrate the termination problem, consider the rules in Figure 23.04. Here, rule R1 is triggered by an INSERT event on TABLE1 and its action includes an update event on ATTRIBUTE1 of TABLE2.

However, rule R2's triggering event is an UPDATE event on ATTRIBUTE1 of TABLE2, and its action includes an INSERT event on TABLE1. It is easy to see in this example that these two rules can trigger one another indefinitely, leading to non-termination. However, if dozens of rules are written, it is very difficult to determine whether termination is guaranteed or not.



```
R1:  CREATE TRIGGER T1
     AFTER INSERT ON TABLE1
     FOR EACH ROW
        UPDATE TABLE2
        SET Attribute1 = . . . ;

R2:  CREATE TRIGGER T2
     AFTER UPDATE OF Attribute1 ON TABLE2
     FOR EACH ROW
        INSERT INTO TABLE1 VALUES ( . . . ) ;
```

Fig: An example to illustrate the termination problem for active rule

If active rules are to reach their potential, it is necessary to develop tools for the design, debugging, and monitoring of active rules that can help users in designing and debugging their rules.

23.1.4 Potential Applications for Active Databases

Finally, we will briefly discuss some of the potential applications of active rules. Obviously, one important application is to allow notification of certain conditions that occur. For example, an active database may be used to monitor, say, the temperature of an industrial furnace. The application can periodically insert in the database the temperature reading records directly from temperature sensors, and active rules can be written that are triggered whenever a temperature record is inserted, with a condition that checks if the temperature exceeds the danger level, and the action to raise an alarm.

Active rules can also be used to enforce integrity constraints by specifying the types of events that may cause the constraints to be violated and then evaluating appropriate conditions that check whether the constraints are actually violated by the event or not. Hence, complex application constraints, often known as business rules may be enforced that way. For example, in the UNIVERSITY database application, one rule may monitor the grade point average of students whenever a new grade is entered, and it may alert the advisor if the GPA of a student falls below a certain threshold; another rule may check that course prerequisites are satisfied before allowing a student to enroll in a course; and so on.

Other applications include the automatic maintenance of derived data, such as the examples of rules R1 through R4 that maintain the derived attribute TOTAL_SAL whenever individual employee tuples are changed. A similar application is to use active rules to maintain the consistency of materialized views whenever the base relations are modified. This application is also relevant to the new data warehousing technologies (see Chapter 26). A related application is to maintain replicated tables consistent by specifying rules that modify the replicas whenever the master table is modified.

23.2 Temporal Database Concepts

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been developed since the early days of database usage. However, in creating these applications, it was mainly left to the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include healthcare, where patient histories need to be maintained; insurance, where claims and accident histories are required as well as information on the times when insurance policies are in effect; reservation systems in general (hotel, airline, car rental, train, etc.), where information on the dates and times when reservations are in effect are required; scientific databases, where data collected from experiments includes the time when each data is measured; and so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee. In the UNIVERSITY database, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE; the grade history of a STUDENT; and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. Users often attempted to simplify or ignore temporal aspects because of the complexity that they add to their applications.

23.2.1 Time Representation, Calendars, and Time Dimensions

For temporal databases, time is considered to be an ordered sequence of points in some granularity that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second in time using this granularity. In reality, each second is a (short) time duration, not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchers have used the term chronon instead of point to describe this minimal granularity for a particular application. The main consequence of choosing a minimum granularity—say, one second—is that events occurring within the same second will be considered to be simultaneous events, even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (Western), Chinese, Islamic, Hindu, Jewish, Coptic, etc.) with different reference points. A calendar organizes time into different time units for convenience. Most calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth's rotation around its axis), and 7 days into a week. Further grouping of days into months and months into years either follow solar or lunar natural phenomena, and are generally irregular. In the Gregorian calendar, which is used in most Western countries, days are grouped into months that are either 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.
In SQL2, the temporal data types include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including sub-second divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an anchored time duration with a fixed starting point, such as the 10-day period from January 1, 1999 to January 10, 1999, inclusive).

23.2.2 Incorporating Time in Relational Databases Using Tuple Versioning

Let us now see how the different types of temporal databases may be represented in the relational model. First, suppose that we would like to include the history of changes as they occur in the real world. Consider again the database in Figure 23.01, and let us assume that, for this application, the granularity is day. Then, we could convert the two relations EMPLOYEE and DEPARTMENT into valid time relations by adding the attributes VST (Valid Start Time) and VET (Valid End Time), whose data type is DATE in order to provide day granularity. This is shown in Figure 23.06(a), where the relations have been renamed EMP_VT and DEPT_VT, respectively.

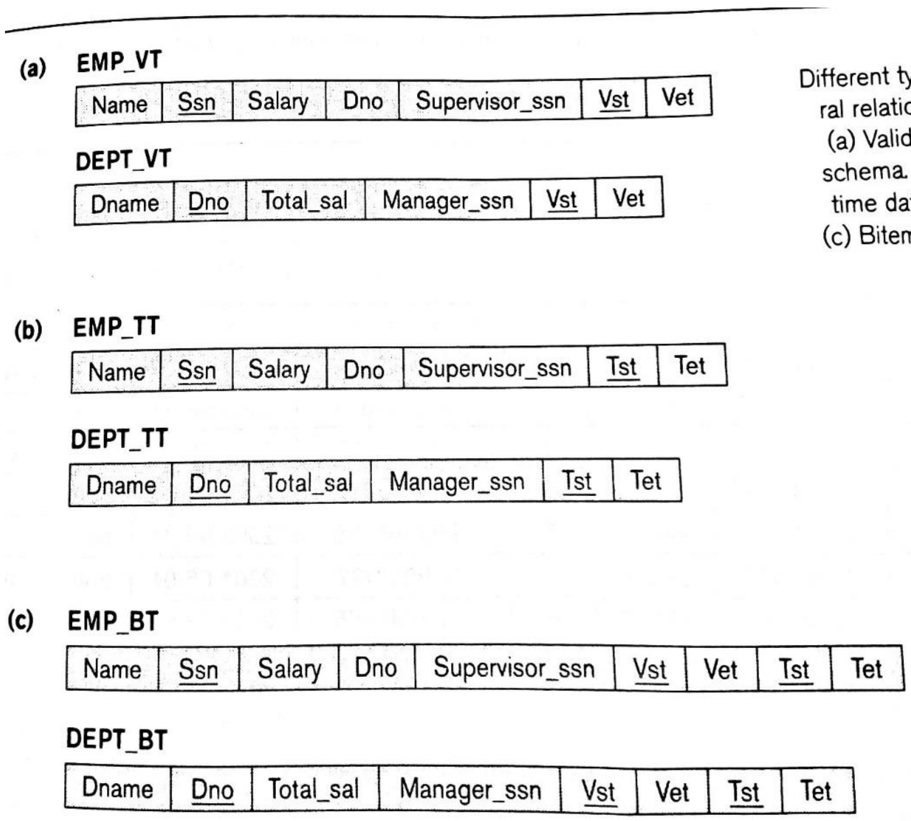
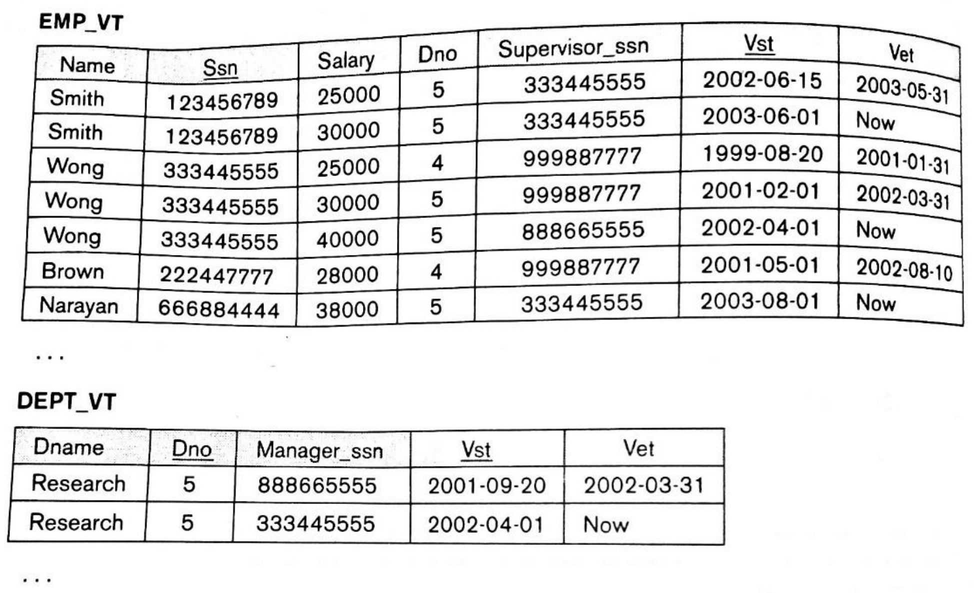

(a)
EMP_VT
Name | Ssn | Salary | Dno | Supervisor_ssn | Vst | Vet
DEPT_VT
Dname | Dno | Total_sal | Manager_ssn | Vst | Vet
(b)
EMP_TT
Name | Ssn | Salary | Dno | Supervisor_ssn | Tst | Tet
DEPT_TT
Dname | Dno | Total_sal | Manager_ssn | Tst | Tet
(c)
EMP_BT
Name | Ssn | Salary | Dno | Supervisor_ssn | Vst | Vet | Tst | Tet
DEPT_BT
Dname | Dno | Total_sal | Manager_ssn | Vst | Vet | Tst | Tet

Figure 24.7
Different types of temporal relational databases. (a) Valid time database schema. (b) Transaction time database schema. (c) Bitemporal database schema.

Consider how the EMP_VT relation differs from the nontemporal EMPLOYEE relation (Figure 23.01). In EMP_VT, each tuple v represents a version of an employee's information that is valid (in the real world) only during the time period [v.VST, v.VET], whereas in EMPLOYEE each tuple represents only the current state or current version of each employee. In EMP_VT, the current version of each employee typically has a special value, **now**, as its valid end time. This special value, now, is a temporal variable that implicitly represents the current time as time progresses. The nontemporal EMPLOYEE relation would only include those tuples from the EMP_VT relation whose VET is now.

Figure 23.07 shows a few tuple versions in the valid-time relations EMP_VT and DEPT_VT. There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan. We can now see how a valid time relation should behave when information is changed. Whenever one or more attributes of an employee are updated, rather than actually overwriting the old values, as would happen in a nontemporal relation, the system should create a new version and close the current version by changing its VET to the end time. Hence, when the user issued the command to update the salary of Smith effective on June 1, 1998 to $30000, the second version of Smith was created (see Figure 23.07). At the time of this update, the first version of Smith was the current version, with now as its VET, but after the update now was changed to May 31, 1998 (one less than June 1, 1998 in day granularity), to indicate that the version has become a closed or history version and that the new (second) version of Smith is now the current one.



EMP_VT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31
Smith	123456789	30000	5	333445555	2003-06-01	Now
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31
Wong	333445555	40000	5	888665555	2002-04-01	Now
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10
Narayan	666884444	38000	5	333445555	2003-08-01	Now

. . .

DEPT_VT

Dname	Dno	Manager_ssn	Vst	Vet
Research	5	888665555	2001-09-20	2002-03-31
Research	5	333445555	2002-04-01	Now

. . .

Figure: Some tuple versions in the valid time relations EMP_VT and DEPT_VT

It is important to note that in a valid time relation, the user must generally provide the valid time of an update. For example, the salary update of Smith may have been entered in the database on May 15, 1998 at 8:52:12am, say, even though the salary change in the real world is effective on June 1, 1998. This is called a **proactive update**, since it is applied to the database before it becomes effective in the real world. If the update was applied to the database after it became effective in the real world, it is called a **retroactive update**. An update that is applied at the same time when it becomes effective is called a **simultaneous update**.

The action that corresponds to deleting an employee in a nontemporal database would typically be applied to a valid time database by closing the current version of the employee being deleted. For example, if Smith leaves the company effective January 19, 1999, then this would be applied by changing VET of the current version of Smith from now to 1999-01-19. In Figure 23.07, there is no current version for Brown, because he presumably left the company on 1997-08-10 and was logically deleted. However, because the database is temporal, the old information on Brown is still there.

The operation to insert a new employee would correspond to creating the first tuple version for that employee, and making it the current version, with the VST being the effective (real world) time when the employee starts work. In Figure 23.07, the tuple on Narayan illustrates this, since the first version has not been updated yet.

Notice that in a valid time relation, the nontemporal key, such as SSN in EMPLOYEE, is no longer unique in each tuple (version). The new relation key for EMP_VT is a combination of the nontemporal key and the valid start time attribute VST, so we use (SSN, VST) as primary key. This is because, at any point in time, there should be at most one valid version of each entity. Hence, the constraint that any two tuple versions representing the same entity should have nonintersecting valid time periods should hold on valid time relations. Notice that if the nontemporal primary key value may change over time, it is important to have a unique surrogate key attribute, whose value never changes for each real world entity, in order to relate together all versions of the same real world entity.

Valid time relations basically keep track of the history of changes as they become effective in the real world. Hence, if all real-world changes are applied, the database keeps a history of the real-world states that are represented. However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual database state at any point in time. If the actual database states are more important to an application, then one should use *transaction time relations*.

Transaction Time Relations

In a transaction time database, whenever a change is applied to the database, the actual timestamp of the transaction that applied the change (insert, delete, or update) is recorded. Such a database is most useful when changes are applied simultaneously in the majority of cases—for example, real-time stock trading or banking transactions. If we convert the nontemporal database of Figure 23.01 into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into transaction time relations by adding the attributes TST (Transaction Start Time) and TET (Transaction End Time), whose data type is typically TIMESTAMP. This is shown in Figure 23.06(b), where the relations have been renamed EMP_TT and DEPT_TT, respectively.

In EMP_TT, each tuple v represents a version of an employee's information that was created at actual time v.TST and was (logically) removed at actual time v.TET (because the information was no longer correct). In EMP_TT, the current version of each employee typically has a special value, uc (Until Changed), as its transaction end time, which indicates that the tuple represents correct information until it is changed by some other transaction (Note 17). A transaction time database has also been called a rollback database (Note 18), because a user can logically roll back to the actual database state at any past point in time t by retrieving all tuple versions v whose transaction time period [v.TST,v.TET] includes time point T.

Bitemporal Relations

Some applications require both valid time and transaction time, leading to bitemporal relations. In our example, Figure 23.06(c) shows how the EMPLOYEE and DEPARTMENT non-temporal relations in Figure 23.01 would appear as bitemporal relations EMP_BT and DEPT_BT, respectively. Figure 23.08 shows a few tuples in these relations. In these tables, tuples whose transaction end time TET is uc are the ones representing currently valid information, whereas tuples whose TET is an absolute timestamp are tuples that were valid until (just before) that timestamp. Hence, the tuples with uc in Figure 23.08 correspond to the valid time tuples in Figure 23.07. The transaction start time attribute TST in each tuple is the timestamp of the transaction that created that tuple.

23.2.3 Incorporating Time in Object-Oriented Databases Using Attribute Versioning

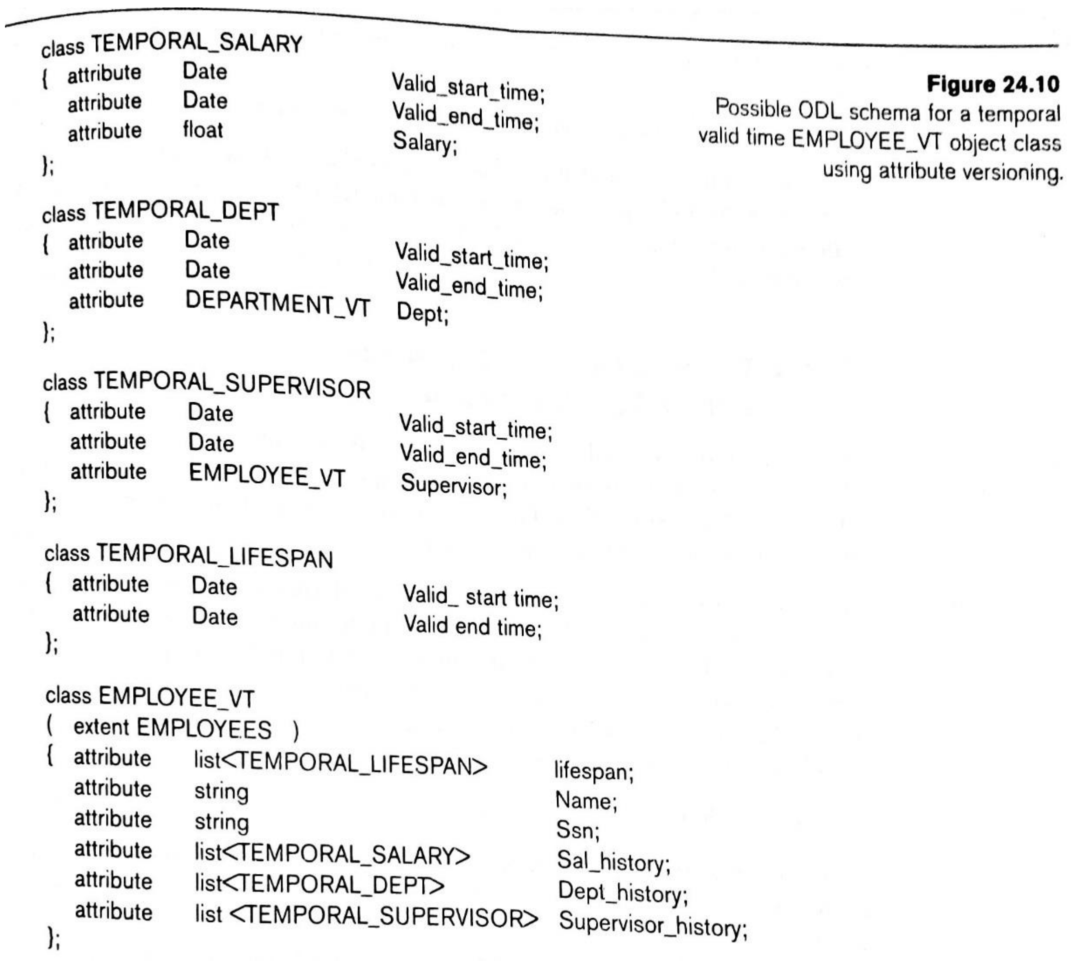
The previous section discussed the tuple versioning approach to implementing temporal databases. In this approach, whenever one attribute value is changed, a whole new tuple version is created, even though all the other attribute values will be identical to the previous tuple version. An alternative approach can be used in database systems that support complex structured objects, such as object databases or object-relational systems. This approach is called attribute versioning.

In attribute versioning, a single complex object is used to store all the temporal changes of the object. Each attribute that changes over time is called a time-varying attribute, and it has its values versioned over time by adding temporal periods to the attribute. The temporal periods may represent valid time, transaction time, or bitemporal, depending on the application requirements. Attributes that do not change are called non-time-varying and are not associated with the temporal periods. To illustrate this, consider the example in Figure 23.09, which is an attribute versioned valid time representation of EMPLOYEE using the ODL notation for object databases (see Chapter 12). Here, we assumed that name and social security number are non-time-varying attributes (they do not change over time), whereas salary, department, and supervisor are time-varying attributes (they may change over time). Each time-varying attribute is represented as a list of tuples *<valid_start_time, valid_end_time, value>*, ordered by valid start time.

Whenever an attribute is changed in this model, the current attribute version is closed and a **new attribute version** for this attribute only is appended to the list. This allows attributes to change asynchronously. The current value for each attribute has now for its *valid_end_time*. When using attribute versioning, it is useful to include a lifespan temporal attribute associated with the whole object whose value is one or more valid time periods that indicate the valid time of existence for the whole object. Logical deletion of the object is implemented by closing the lifespan. The constraint that any time period of an attribute within an object should be a subset of the object's lifespan should be enforced.
For bitemporal databases, each attribute version would have a tuple with five components:

<valid_start_time, valid_end_time, trans_start_time, trans_end_time, value>

The object lifespan would also include both valid and transaction time dimensions. The full capabilities of bitemporal databases can hence be available with attribute versioning. Mechanisms similar to those discussed earlier for updating tuple versions can be applied to updating attribute versions.



```
class TEMPORAL_SALARY
{   attribute    Date              Valid_start_time;
    attribute    Date              Valid_end_time;
    attribute    float             Salary;
};

class TEMPORAL_DEPT
{   attribute    Date              Valid_start_time;
    attribute    Date              Valid_end_time;
    attribute    DEPARTMENT_VT     Dept;
};

class TEMPORAL_SUPERVISOR
{   attribute    Date              Valid_start_time;
    attribute    Date              Valid_end_time;
    attribute    EMPLOYEE_VT       Supervisor;
};

class TEMPORAL_LIFESPAN
{   attribute    Date              Valid_ start time;
    attribute    Date              Valid end time;
};

class EMPLOYEE_VT
(   extent EMPLOYEES   )
{   attribute    list<TEMPORAL_LIFESPAN>       lifespan;
    attribute    string                        Name;
    attribute    string                        Ssn;
    attribute    list<TEMPORAL_SALARY>         Sal_history;
    attribute    list<TEMPORAL_DEPT>           Dept_history;
    attribute    list <TEMPORAL_SUPERVISOR>    Supervisor_history;
};
```

Figure 24.10
Possible ODL schema for a temporal valid time EMPLOYEE_VT object class using attribute versioning.

23.2.5 Time Series Data

Time series data are used very often in financial, sales, and economics applications. They involve data values that are recorded according to a specific predefined sequence of time points. They are hence a special type of **valid event data**, where the event time points are predetermined according to a fixed calendar. Consider the example of closing daily stock prices of a particular company on the New York Stock Exchange. The granularity here is day, but the days that the stock market is open are known (nonholiday weekdays). Hence, it has been common to specify a computational procedure that calculates the particular **calendar** associated with a time series. Typical queries on time series involve **temporal aggregation** over higher granularity intervals—for example, finding the average or maximum weekly closing stock price or the maximum and minimum *monthly* closing stock price from the *daily* information.

As another example, consider the daily sales dollar amount at each store of a chain of stores owned by a particular company. Again, typical temporal aggregates would be retrieving the weekly, monthly, or yearly sales from the daily sales information (using the sum aggregate function), or comparing same store monthly sales with previous monthly sales, and so on.

Because of the specialized nature of time series data, and the lack of support in older DBMSs, it has been common to use specialized time series management systems rather that general purpose DBMSs for managing such information. In such systems, it has been common to store time series values in sequential order in a file, and apply specialized time series procedures to analyze the information. The problem with this approach is that the full power of high-level querying in languages such as SQL will not be available in such systems.

More recently, some commercial DBMS packages are offering time series extensions, such as the time series datablade of Informix Universal Server. In addition, the TSQL2 language provides some support for time series in the form of event tables.

23.3 Spatial and Multimedia Databases

23.3.1 Introduction to Spatial Database Concepts

Spatial databases provide concepts for databases that keep track of objects in a multi-dimensional space. For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects—from countries and states to rivers, cities, roads, seas, and so on. These databases are used in many applications, such as environmental, emergency, and battle management. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points. In general, a spatial database stores objects that have spatial characteristics that describe them. The spatial relationships among the objects are important, and they are often needed when querying the database. Although a spatial database can in general refer to an n-dimensional space for any n, we will limit our discussion to two dimensions as an illustration.

The main extensions that are needed for spatial databases are models that can interpret spatial characteristics. In addition, special indexing and storage structures are often needed to improve performance. Let us first discuss some of the model extensions for two-dimensional spatial databases. The basic extensions needed are to include two-dimensional geometric concepts, such as points, lines and line segments, circles, polygons, and arcs, in order to specify the spatial characteristics of objects. In addition, spatial operations are needed to operate on the objects' spatial characteristics—for example, to compute the distance between two objects—as well as spatial Boolean conditions—for example, to check whether two objects spatially overlap. To illustrate, consider a database that is used for emergency management applications. A description of the spatial positions of many types of objects would be needed. Some of these objects generally have static spatial characteristics, such as streets and highways, water pumps (for fire control), police stations, fire stations, and hospitals. Other objects have dynamic spatial characteristics that change over time, such as police vehicles, ambulances, or fire trucks.

The following categories illustrate three typical types of spatial queries:

- Range query: Finds the objects of a particular type that are within a given spatial area or within a particular distance from a given location. (For example, finds all hospitals within the Dallas city area, or finds all ambulances within five miles of an accident location.)

- Nearest neighbor query: Finds an object of a particular type that is closest to a given location. (For example, finds the police car that is closest to a particular location.)

- Spatial joins or overlays: Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another. (For example, finds all cities that fall on a major highway or finds all homes that are within two miles of a lake.)

For these and other types of spatial queries to be answered efficiently, special techniques for spatial indexing are needed. One of the best known techniques is the use of R-trees and their variations. **R-trees** group together objects that are in close spatial physical proximity on the same leaf nodes of a tree-structured index. Since a leaf node can point to only a certain number of objects, algorithms for dividing the space into rectangular subspaces that include the objects are needed. Typical criteria for dividing the space include minimizing the rectangle areas, since this would lead to a quicker narrowing of the search space. Problems such as having objects with overlapping spatial areas are handled in different ways by the many different variations of R-trees. The internal nodes of R-trees are associated with rectangles whose area covers all the rectangles in its subtree. Hence, R-trees can easily answer queries, such as find all objects in a given area by limiting the tree search to those subtrees whose rectangles intersect with the area given in the query.

Other spatial storage structures include quadtrees and their variations. **Quadtrees** generally divide each space or subspace into equally sized areas, and proceed with the sub-divisions of each subspace to identify the positions of various objects. Recently, many newer spatial access structures have been proposed, and this area is still an active research area.

23.3.2 Introduction to Multimedia Database Concepts

Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures or drawings), video clips (such as movies, newsreels, or home videos), audio clips (such as songs, phone messages, or speeches), and documents (such as books or articles). The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. For example, one may want to locate all video clips in a video database that include a certain person in them, say Bill Clinton. One may also want to retrieve video clips based on certain activities included in them, such as a video clips were a goal is scored in a soccer game by a certain player or team.

The above types of queries are referred to as **content-based retrieval**, because the multimedia source is being retrieved based on its containing certain objects or activities. Hence, a multimedia database must use some model to organize and index the multimedia sources based on their contents. Identifying the contents of multimedia sources is a difficult and time-consuming task. There are two main approaches. The first is based on **automatic analysis** of the multimedia sources to identify certain mathematical characteristics of their contents. This approach uses different techniques depending on the type of multimedia source (image, text, video, or audio). The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all the different multimedia sources, but it requires a manual preprocessing phase where a person has to scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index these sources.

Characteristics of each type of multimedia source:

Image:
An image is typically stored either in raw form as a set of pixel or cell values, or in compressed form to save space. The image shape descriptor describes the geometric shape of the raw image, which is typically a rectangle of cells of a certain width and height. Hence, each image can be represented by an m by n grid of cells. Each cell contains a pixel value that describes the cell content. In black/white images, pixels can be one bit. In gray scale or color images, a pixel is multiple bits. Because images may require large amounts of space, they are often stored in compressed form. Compression standards, such as the GIF standard, use various mathematical transformations to reduce the number of cells stored but still maintain the main image characteristics. The mathematical transforms that can be used include Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and wavelet transforms.

To identify objects of interest in an image, the image is typically divided into homogeneous segments using a homogeneity predicate. For example, in a color image, cells that are adjacent to one another and whose pixel values are close are grouped into a segment. The homogeneity predicate defines the conditions for how to automatically group those cells. Segmentation and compression can hence identify the main characteristics of an image.

A typical image database query would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern. There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group together stored images that are close in the distance metric so as to limit the search space. The second approach, called the **transformation approach**, measures image similarity by having a small number of transformations that can transform one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the latter approach is more general, it is also more time consuming and difficult.

Video:
A video source is typically represented as a sequence of frames, where each frame is a still image. However, rather than identifying the objects and activities in every individual frame, the video is divided into video segments, where each segment is made up of a sequence of contiguous frames that includes the same objects/activities. Each segment is identified by its starting and ending frames. The objects and activities identified in each video segment can be used to index the segments. An indexing technique called frame segment trees has been proposed for video indexing. The index includes both objects, such as persons, houses, cars, and activities, such as a person delivering a speech or two people talking.

Text:
A text/document source is basically the full text of some article, book, or magazine. These sources are typically indexed by identifying the keywords that appear in the text and their relative frequencies. However, filler words are eliminated from that process. Because there could be too many keywords when attempting to index a collection of documents, techniques have been developed to reduce the number of keywords to those that are most relevant to the collection. A technique called singular value decompositions (SVD), which is based on matrix transformations, can be used for this purpose. An indexing technique called telescoping vector trees, or TV-trees, can then be used to group similar documents together.

Audio:
Audio sources include stored recorded messages, such as speeches, class presentations, or even surveillance recording of phone messages or conversations by law enforcement. Here, discrete transforms can be used to identify the main characteristics of a certain person's voice in order to have similarity based indexing and retrieval. Audio characteristic features include loudness, intensity, pitch, and clarity.

25.1 Introduction to Deductive Databases

In a deductive database system, we typically specify rules through a declarative language—a language in which we specify what to achieve rather than how to achieve it. An inference engine (or deduction mechanism) within the system can deduce new facts from the database by interpreting these rules. The model used for deductive databases is closely related to the relational data model, and particularly to the domain relational calculus formalism (see Section 9.4). It is also related to the field of logic programming and the Prolog language. The deductive database work based on logic has used Prolog as a starting point. A variation of Prolog called Datalog is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language. Although the language structure of Datalog resembles that of Prolog, its operational semantics—that is, how a Datalog program is to be executed—is still a topic of active research.

A deductive database uses two main types of specifications: facts and rules. Facts are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names. Recall that a tuple in a relation describes some real-world fact whose meaning is partly determined by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is determined solely by its position within the tuple. Rules are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications. The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of standard relational views.

The evaluation of Prolog programs is based on a technique called backward chaining, which involves a top-down evaluation of goals. In the deductive databases that use Datalog, attention has been devoted to handling large volumes of data stored in a relational database. Hence, evaluation techniques have been devised that resemble those for a bottom-up evaluation. Prolog suffers from the limitation that the order of specification of facts and rules is significant in evaluation; moreover, the order of literals within a rule is significant. The execution techniques for Datalog programs attempt to circumvent these problems.

Object-oriented databases (OODBs) is instructive to put deductive databases (DDBs) in a proper context with respect to OODBs. The emphasis in OODBs has been on providing a natural modeling mechanism for real-world objects by encapsulating their structure with behavior. The emphasis in DDBs, in contrast, has been on deriving new knowledge from existing data by supplying additional real-world relationships in the form of rules. A marriage of these two different enhancements to traditional databases has started appearing, in the form of adding deductive capabilities to OODBs and adding programming language interfaces like C++ to DDBs. This new breed of databases systems is referred to as deductive OODBs, or DOODs for short.

25.2 Prolog/Datalog Notation

The notation used in Prolog/Datalog is based on providing predicates with unique names. A **predicate** has an implicit meaning, which is suggested by the predicate name, and a fixed number of arguments. If the arguments are all constant values, the predicate simply states that a certain fact is true. If, on the other hand, the predicate has variables as arguments, it is either considered as a query or as part of a rule or constraint. Throughout this chapter, we adopt the Prolog convention that all constant values in a predicate are either numeric or character strings; they are represented as identifiers (or names) starting with lowercase letters only, whereas variable names always start with an uppercase letter.

25.2.1 An Example

Consider the example shown in Figure 25.01, which is based on the relational database of Figure 07.06, but in a much simplified form. There are three predicate names: supervise, superior, and subordinate. The supervise predicate is defined via a set of facts, each of which has two arguments: a supervisor name, followed by the name of a direct supervisee (subordinate) of that supervisor. These facts correspond to the actual data that is stored in the database, and they can be considered as constituting a set of tuples in a relation SUPERVISE with two attributes whose schema is

SUPERVISE(Supervisor,Supervisee)

Thus, supervise(X, Y) states the fact that "X supervises Y." Notice the omission of the attribute names in the Prolog notation. Attribute names are only represented by virtue of the position of each argument in a predicate: the first argument represents the supervisor, and the second argument represents a direct subordinate.

The other two predicate names are defined by rules. The main contribution of deductive databases is the ability to specify recursive rules, and to provide a framework for inferring new information based on the specified rules. A rule is of the form head :- body, where :- is read as "if and only if". A rule usually has a single predicate to the left of the :- symbol—called the head or left-hand side (LHS) or conclusion of the rule—and one or more predicates to the right of the :- symbol—called the body or right-hand side (RHS) or premise(s) of the rule. A predicate with constants as arguments is said to be ground; we also refer to it as an instantiated predicate. The arguments of the predicates that appear in a rule typically include a number of variable symbols, although predicates can also contain constants as arguments. A rule specifies that, if a particular assignment or binding of constant values to the variables in the body (RHS predicates) makes all the RHS predicates true, it also makes the head (LHS predicate) true by using the same assignment of constant values to variables. Hence, a rule provides us with a way of generating new facts that are instantiations of the head of the rule. These new facts are based on facts that already exist, corresponding to the instantiations (or bindings) of predicates in the body of the rule. Notice that by listing multiple predicates in the body of a rule we implicitly apply the logical and operator to these predicates. Hence, the commas between the RHS predicates may be read as meaning "and."

Consider the definition of the predicate superior in Figure 25.01, whose first argument is an employee name and whose second argument is an employee who is either a direct or an indirect subordinate of the first employee. By indirect subordinate, we mean the subordinate of some subordinate down to any number of levels. Thus superior(X, Y) stands for the fact that "X is a superior of Y" through direct or indirect supervision. We can write two rules that together specify the meaning of the new predicate. The first rule under Rules in the figure states that, for every value of X and Y, if supervise(X, Y)—the rule body—is true, then superior(X, Y)—the rule head—is also true, since Y would be a direct subordinate of X (at one level down). This rule can be used to generate all direct superior/subordinate relationships from the facts that define the supervise predicate. The second recursive rule states that, if supervise(X, Z) and superior(Z, Y) are both true, then superior(X, Y) is also true. This is an example of a recursive rule, where one of the rule body predicates in the RHS is the same as the rule head predicate in the LHS. In general, the rule body defines a number of premises such that, if they are all true, we can deduce that the conclusion in the rule head is also true. Notice that, if we have two (or more) rules with the same head (LHS predicate), it is equivalent to saying that the predicate is true (that is, that it can be instantiated) if either one of the bodies is true; hence, it is equivalent to a logical or operation.

A Prolog system contains a number of built-in predicates that the system can interpret directly. These typically include the equality comparison operator =(X,Y), which returns true if X and Y are identical and can also be written as X=Y by using the standard infix notation. Other comparison operators for numbers, such as <, <=, >, and >=, can be treated as binary predicates. Arithmetic functions such as +, -, *, and / can be used as arguments in predicates in Prolog. In contrast, Datalog (in its basic form) does not allow functions such as arithmetic operations as arguments; indeed, this is one of the main differences between Prolog and Datalog. However, later extensions to Datalog have been proposed to include functions.

(a) **Facts**

```
SUPERVISE(franklin, john).
SUPERVISE(franklin, ramesh).
SUPERVISE(franklin, joyce).
SUPERVISE(jennifer, alicia).
SUPERVISE(jennifer, ahmad).
SUPERVISE(james, franklin).
SUPERVISE(james, jennifer).
...
```

Rules

```
SUPERIOR(X, Y) :- SUPERVISE(X, Y).
SUPERIOR(X, Y) :- SUPERVISE(X, Z), SUPERIOR(Z, Y).
SUBORDINATE(X, Y) :- SUPERIOR(Y, X).
```

Queries

```
SUPERIOR(james, Y)?
SUPERIOR(james, joyce)?
```

(b)

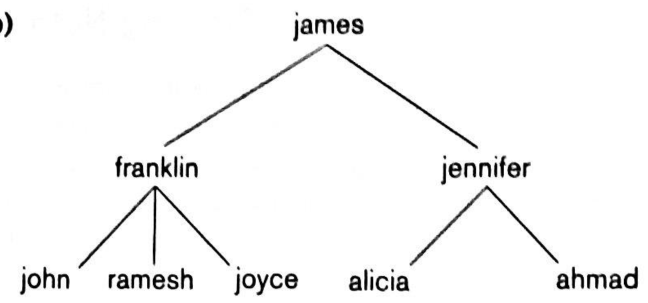


Figure 24.11
(a) Prolog notation. (b) The supervisory tree.

A query typically involves a predicate symbol with some variable arguments, and its meaning (or "answer") is to deduce all the different constant combinations that, when bound (assigned) to the variables, can make the predicate true. For example, the first query in Figure 25.01 requests the names of all subordinates of "james" at any level. A different type of query, which has only constant symbols as arguments, returns either a true or a false result, depending on whether the arguments provided can be deduced from the facts and rules. For example, the second query in Figure 25.01 returns true, since superior(james, joyce) can be deduced.

25.2.2 Datalog Notation

In Datalog, as in other logic-based languages, a program is built from basic objects called atomic formulas. It is customary to define the syntax of logic-based languages by describing the syntax of atomic formulas and identifying how they can be combined to form a program. In Datalog, atomic formulas are literals of the form p(a1, a2, ..., an), where p is the predicate name and n is the number of arguments for predicate p. Different predicate symbols can have different numbers of arguments, and the number of arguments n of predicate p is sometimes called the arity or degree of p. The arguments can be either constant values or variable names. As mentioned earlier, we use the convention that constant values either are numeric or start with a lowercase character, whereas variable names always start with an uppercase character.

A number of built-in predicates are included in Datalog, which can also be used to construct atomic formulas. The built-in predicates are of two main types: the binary comparison predicates <(less), <=(less_or_equal), >(greater), and >= (greater_or_equal) over ordered domains; and the comparison predicates = (equal) and /= (not_equal) over ordered or unordered domains. These can be used as binary predicates with the same functional syntax as other predicates—for example by writing less(X, 3)—or they can be specified by using the customary infix notation

X<3. Notice that, because the domains of these predicates are potentially infinite, they should be used with care in rule definitions. For example, the predicate greater(X, 3), if used alone, generates an infinite set of values for X that satisfy the predicate (all integer numbers greater than 3).

A **literal** is either an atomic formula as defined earlier—called a positive literal—or an atomic formula preceded by not. The latter is a negated atomic formula, called a negative literal. Datalog programs can be considered to be a subset of the predicate calculus formulas, which are somewhat similar to the formulas of the domain relational calculus. In Datalog, however, these formulas are first converted into what is known as clausal form before they are expressed in Datalog; and only formulas given in a restricted clausal form, called Horn clauses, can be used in Datalog.

25.2.3 Clausal Form and Horn Clauses

Aformula in the relational calculus is a condition that includes predicates called atoms (based on relation names). In addition, a formula can have quantifiers—namely, the universal quantifier (for all) and the existential quantifier (there exists). In clausal form, a formula must be transformed into another formula with the following characteristics:

- All variables in the formula are universally quantified. Hence, it is not necessary to include the universal quantifiers (for all) explicitly; the quantifiers are removed, and all variables in the formula are implicitly quantified by the universal quantifier.

- In clausal form, the formula is made up of a number of clauses, where each clause is composed of a number of literals connected by OR logical connectives only. Hence, each clause is a disjunction of literals.

- The clauses themselves are connected by AND logical connectives only, to form a formula.

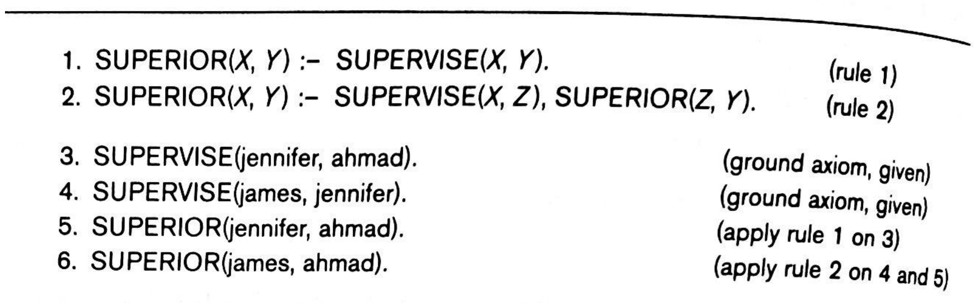
Hence, the clausal form of a formula is a conjunction of clauses.

It can be shown that any formula can be converted into clausal form. For our purposes, we are mainly interested in the form of the individual clauses, each of which is a disjunction of literals. Recall that literals can be positive literals or negative literals.

25.3 Interpretations of Rules

There are two main alternatives for interpreting the theoretical meaning of rules: proof-theoretic and model-theoretic. In practical systems, the inference mechanism within a system defines the exact interpretation, which may not coincide with either of the two theoretical interpretations. The inference mechanisms are then discussed briefly as a way of defining the meaning of rules.

In the proof-theoretic interpretation of rules, we consider the facts and rules to be true statements, or axioms. Ground axioms contain no variables. The facts are ground axioms that are given to be true. Rules are called deductive axioms, since they can be used to deduce new facts. The deductive axioms can be used to construct proofs that derive new facts from existing facts. For example, Figure 25.02 shows how to prove the fact superior(james, ahmad) from the rules and facts given in Figure 25.01. The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query. The process of proving whether a certain fact (theorem) holds is known as *theorem proving*.



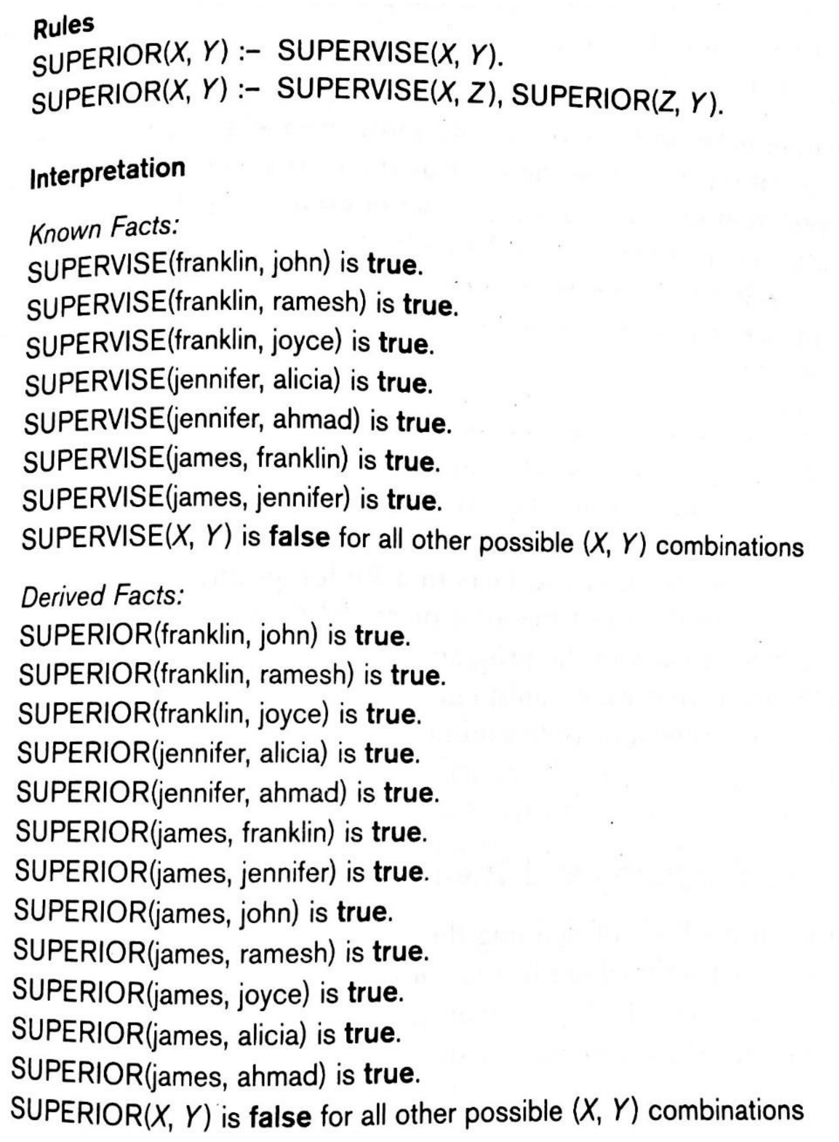
```
1. SUPERIOR(X, Y) :-  SUPERVISE(X, Y).                    (rule 1)
2. SUPERIOR(X, Y) :-  SUPERVISE(X, Z), SUPERIOR(Z, Y).    (rule 2)

3. SUPERVISE(jennifer, ahmad).             (ground axiom, given)
4. SUPERVISE(james, jennifer).             (ground axiom, given)
5. SUPERIOR(jennifer, ahmad).              (apply rule 1 on 3)
6. SUPERIOR(james, ahmad).                 (apply rule 2 on 4 and 5)
```

Figure: Providing a new fact

The second type of interpretation is called the model-theoretic interpretation. Here, given a finite or an infinite domain of constant values (Note 4), we assign to a predicate every possible combination of values as arguments. We must then determine whether the predicate is true or false. In general, it is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an interpretation of the set of predicates. For example, consider the interpretation shown in Figure 25.03 for the predicates supervise and superior. This interpretation assigns a truth value (true or false) to every possible combination of argument values (from a finite domain) for the two predicates.

An interpretation is called a model for a specific set of rules if those rules are always true under that interpretation; that is, for any values assigned to the variables in the rules, the head of the rules is true when we substitute the truth values assigned to the predicates in the body of the rule by that interpretation. Hence, whenever a particular substitution (binding) to the variables in the rules is applied, if all the predicates in the body of a rule are true under the interpretation, the predicate in the head of the rule must also be true. The interpretation shown in Figure 25.03 is a model for the two rules shown, since it can never cause the rules to be violated. Notice that a rule is violated if a particular binding of constants to the variables makes all the predicates in the rule body true but makes the predicate in the rule head false. For example, if supervise(a, b) and superior(b, c) are both true under some interpretation, but superior(a, c) is not true, the interpretation cannot be a model for the recursive rule:



Rules

SUPERIOR(X, Y) :- SUPERVISE(X, Y).
SUPERIOR(X, Y) :- SUPERVISE(X, Z), SUPERIOR(Z, Y).

Interpretation

Known Facts:
SUPERVISE(franklin, john) is **true**.
SUPERVISE(franklin, ramesh) is **true**.
SUPERVISE(franklin, joyce) is **true**.
SUPERVISE(jennifer, alicia) is **true**.
SUPERVISE(jennifer, ahmad) is **true**.
SUPERVISE(james, franklin) is **true**.
SUPERVISE(james, jennifer) is **true**.
SUPERVISE(X, Y) is **false** for all other possible (X, Y) combinations

Derived Facts:
SUPERIOR(franklin, john) is **true**.
SUPERIOR(franklin, ramesh) is **true**.
SUPERIOR(franklin, joyce) is **true**.
SUPERIOR(jennifer, alicia) is **true**.
SUPERIOR(jennifer, ahmad) is **true**.
SUPERIOR(james, franklin) is **true**.
SUPERIOR(james, jennifer) is **true**.
SUPERIOR(james, john) is **true**.
SUPERIOR(james, ramesh) is **true**.
SUPERIOR(james, joyce) is **true**.
SUPERIOR(james, alicia) is **true**.
SUPERIOR(james, ahmad) is **true**.
SUPERIOR(X, Y) is **false** for all other possible (X, Y) combinations

Figure: An interpretation

27.3 Mobile Databases

Recent advances in wireless technology have led to mobile computing, a new dimension in data communication and processing. The mobile computing environment will provide database applications with useful aspects of wireless technology. The mobile computing platform allows users to establish communication with other users and to manage their work while they are mobile. This feature is especially useful to geographically dispersed organizations. Typical examples might include traffic olice, taxi dispatchers, and weather reporting services, as well as financial market reporting and information brokering applications. However, there are a number of hardware as well as software problems that must be resolved before the capabilities of mobile computing can be fully utilized. Some of the software problems—which may involve data management, transaction management, and database recovery—have their origin in distributed database systems. In mobile computing, however, these problems become more difficult to solve, mainly because of the narrow bandwidth of the wireless communication channels, the relatively short active life of the power supply (battery) of mobile units, and the changing locations of required information (sometimes in cache, sometimes in the air, sometimes at the server). In addition, mobile computing has its own unique architectural challenges.

27.3.1 Mobile Computing Architecture

The general architecture of a mobile platform is illustrated in Figure 27.04. It is a distributed architecture where a number of computers, generally referred to as Fixed Hosts (FS) and Base Stations (BS), are interconnected through a high-speed wired network. Fixed hosts are general purpose computers that are not equipped to manage mobile units but can be configured to do so. Base stations are equipped with wireless interfaces and can communicate with mobile units to support data access.

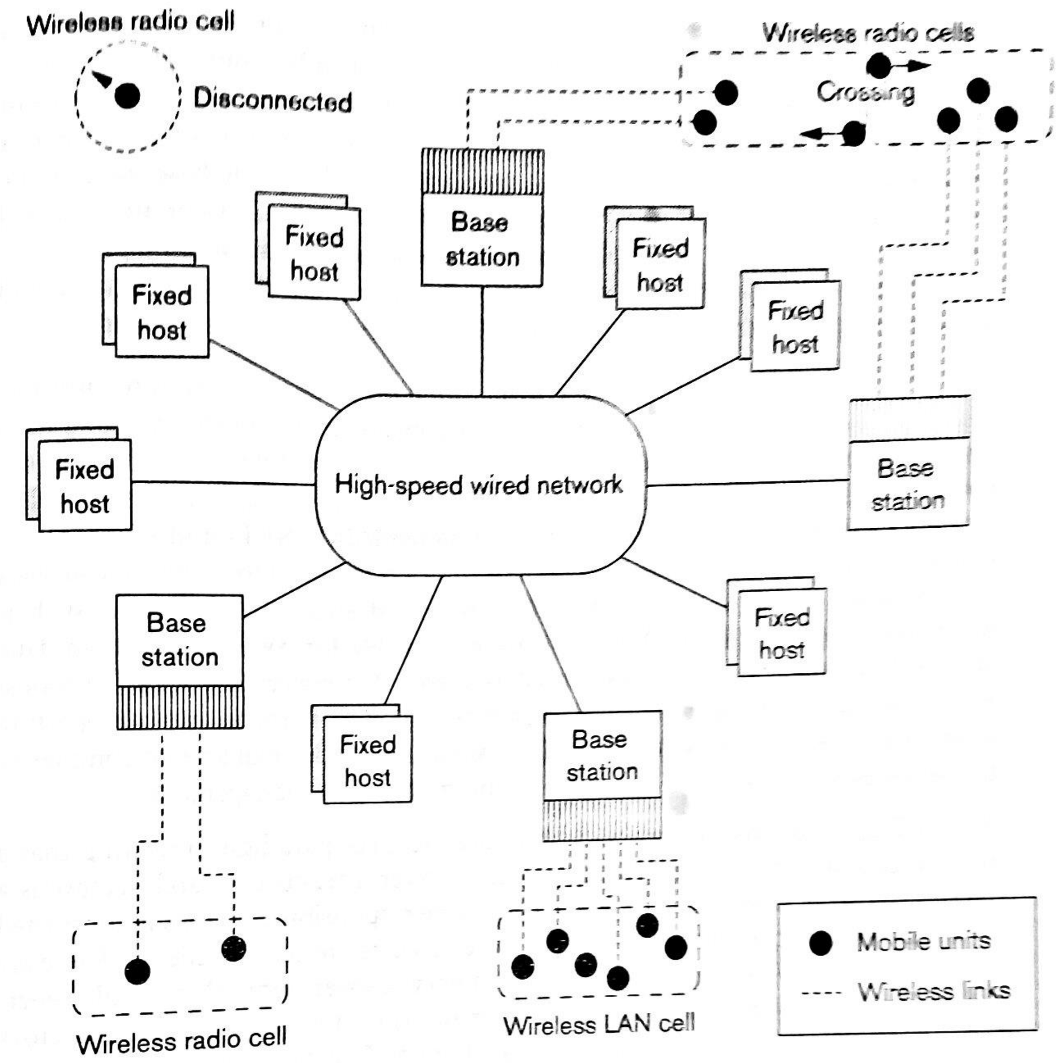


Figure: A general architecture of an infrastructure- based mobile platform

Mobile Units (MU) (or hosts) and base stations communicate through wireless channels having bandwidths significantly lower than those of a wired network. **A downlink channel** is used for sending data from a BS to an MU and an **uplink channel** is used for sending data from an MU to its BS. Recent products for portable wireless have an upper limit of 1 Mbps (megabits per second) for infrared communication, 2 Mbps for radio communication, and 9.14 Kbps (kilobits per second) for cellular telephony. Ethernet, by comparison, provides 10 Mbps fast Ethernet and FDDI provide 100 Mbps and ATM (asynchronous transfer mode) provides 155 Mbps.

Mobile units are battery-powered portable computers that move freely in a **geographic mobility domain**, an area that is restricted by the limited bandwidth of wireless communication channels. To manage the mobility of units, the entire geographic mobility domain is divided into smaller domains called **cells**. The mobile discipline requires that the movement of mobile units be unrestricted within the geographic mobility domain (intercell movement), while having information access contiguity during movement guarantees that the movement of a mobile unit across cell boundaries will have no effect on the data retrieval process.

The mobile computing platform can be effectively described under the client-server paradigm, which means we may sometimes refer to a mobile unit as a client or sometimes as a user, and the base stations as servers. Each cell is managed by a base station, which contains transmitters and receivers for responding to the information-processing needs of clients located in the cell. We assume that the average query response time is much shorter than the time required by the client to physically traverse a cell.

Clients and servers communicate through wireless channels. The communication link between a client and a server may be modeled as multiple data channels or as a single channel.

Characteristics of Mobile Environments
In mobile database environments, data generally changes very rapidly. Users often query servers to remain up-to-date. More specifically, they often want to track every broadcast for their data item of interest. Examples of this type of data are stock market information, weather data, and airline information. The database is updated asynchronously by an independent external process.

Users are mobile and randomly enter and exit from cells. The average duration of a user's stay in the cell is referred to as **residence latency (RL)**, a parameter that is computed (and continually adjusted) by observing user residence times in cells. Thus each cell has an RL value. User reference behavior tends to be localized—i.e., users tend to access certain parts of the database very frequently. Servers maintain neither client arrival and departure patterns nor client-specific data request information.

Wireless networks differ from wired networks in many ways. Database users over a wired network remain connected not only to the network but also to a continuous power source. Thus, response time is the key performance metric. In a wireless network, however, both the response time and the active life of the user's power source (battery) are important. While a mobile unit is listening or transmitting on-line, it is considered to be in active mode. For current laptops with CD-ROM drives, estimated battery life in active mode is under 3 hours. In order to conserve energy and extend battery life, clients can slip into a doze mode, where they are not actively listening on the channel and they can expend significantly less energy than they do in active mode. Clients can be woken up from the doze mode when the server needs to communicate with the client.

27.3.3 Data Management Issues

From a data management standpoint, mobile computing may be considered a variation of distributed computing. Mobile databases can be distributed under two possible scenarios:

1. The entire database is distributed mainly among the wired components, possibly with full or partial replication. A base station manages its own database with a DBMS-like functionality, with additional functionality for locating mobile units and additional query and transaction management features to meet the requirements of mobile environments.

2. The database is distributed among wired and wireless components. Data management responsibility is shared among base stations and mobile units.

It can also be applied to mobile databases with the following additional considerations and variations: 4

1. *Data distribution and replication*: Data is unevenly distributed among the base stations and mobile units. The consistency constraints compound the problem of cache management. Caches attempt to provide the most frequently accessed and updated data to mobile units that process their own transactions and may be disconnected over long periods.

2. *Transaction models*: Issues of fault tolerance and correctness of transactions are aggravated in the mobile environment. A mobile transaction is executed sequentially through several base stations and possibly on multiple data sets depending upon the movement of the mobile unit. Central coordination of transaction execution is lacking, particularly in scenario (2) above. Hence, traditional ACID properties of transactions may need to be modified and new transaction models must be defined.

3. *Query processing*: Awareness of where the data is located is important and affects the cost/benefit analysis of query processing. The query response needs to be returned to mobile units that may be in transit or may cross cell boundaries yet must receive complete and correct query results.

4. *Recovery and fault tolerance*: The mobile database environment must deal with site, media, transaction, and communication failures. Site failure at an MU is frequently due to limited battery power. If an MU has a voluntary shutdown, it should not be treated as a failure. Transaction failures are more frequent during handoff when an MU crosses cells. MU failure causes a network partitioning and affects routing algorithms.

5. *Mobile database design:* The global name resolution problem for handling queries is compounded because of mobility and frequent shutdown. Mobile database design must consider many issues of metadata management—for example, the constant updating of location information.

27.4 Geographic Information Systems

Geographic information systems (GIS) are used to collect, model, store, and analyze information describing physical properties of the geographical world. The scope of GIS broadly encompasses two types of data:

(1) spatial data, originating from maps, digital images, administrative and political boundaries, roads, transportation networks; physical data such as rivers, soil characteristics, climatic regions, land elevations, and

(2) nonspatial data, such as census counts, economic data, and sales or marketing information. GIS is a rapidly developing domain that offers highly innovative approaches to meet some challenging technical demands.

27.4.1 GIS Applications

It is possible to divide GISs into three categories:
(1) Cartographic applications
(2) Digital terrain modeling applications, and
(3) Geographic objects applications.

In cartographic and terrain modeling applications, variations in spatial attributes are captured—for example, soil characteristics, crop density, and air quality. In geographic objects applications, objects of interest are identified from a physical domain—for example, power plants, electoral districts, property parcels, product distribution districts, and city landmarks. These objects are related with pertinent application data—which may be, for this specific example, power consumption, voting patterns, property sales volumes, product sales volume, and traffic density.

The first two categories of GIS applications require a field-based representation, whereas the third category requires an object-based one. The cartographic approach involves special functions that can include the overlapping of layers of maps to combine attribute data that will allow, for example, the measuring of distances in three-dimensional space and the reclassification of data on the map. Digital terrain modeling requires a digital representation of parts of earth's surface using land elevations at sample points that are connected to yield a surface model such as a three-dimensional net (connected lines in 3D) showing the surface terrain. It requires functions of interpolation between observed points as well as visualization. In object-based geographic applications, additional spatial functions are needed to deal with data related to roads, physical pipelines, communication cables, power lines, and such. For example, for a given region, comparable maps can be used for comparison at various points of time to show changes in certain data such as locations of roads, cables, buildings, and streams.

27.4.2 Data Management Requirements of GIS

Data Modeling and Representation

GIS data can be broadly represented in two formats: (1) vector and (2) raster. Vector data represents geometric objects such as points, lines, and polygons. Thus a lake may be represented as a polygon, a river by a series of line segments. Raster data is characterized as an array of points, where each point represents the value of an attribute for a real-world location. Informally, raster images are n-dimensional arrays where each entry is a unit of the image and represents an attribute. Two-dimensional units are called pixels, while three-dimensional units are called voxels. Three-dimensional elevation data is stored in a raster-based **digital elevation model (DEM)** format. Another raster format called **triangular irregular network (TIN)** is a topological vector-based approach that models surfaces by connecting sample points as vertices of triangles and has a point density that may vary with the roughness of the terrain. Rectangular grids (or elevation matrices) are two-dimensional array structures. In **digital terrain modeling (DTM),** the model also may be used by substituting the elevation with some attribute of interest such as population density or air temperature. GIS data often includes a temporal structure in addition to a spatial structure. For example, traffic density may be measured every 60 seconds at a set of points.

Data Analysis

GIS data undergoes various types of analysis. For example, in applications such as soil erosion studies, environmental impact studies, or hydrological runoff simulations, DTM data may undergo various types of **geomorphometric analysis**—measurements such as slope values, gradients (the rate of change in altitude), aspect (the compass direction of the gradient), profile convexity (the rate of change of gradient), plan convexity (the convexity of contours and other parameters). When GIS data is used for decision support applications, it may undergo aggregation and expansion operations using data warehousing, as we discussed in Section 26.1.5. In addition, geometric operations (to compute distances, areas, volumes), topological operations (to compute overlaps, intersections, shortest paths), and temporal operations (to compute internal-based or event-based queries) are involved. Analysis involves a number of temporal and spatial operations.

Data Integration

GISs must integrate both vector and raster data from a variety of sources. Sometimes edges and regions are inferred from a raster image to form a vector model, or conversely, raster images such as aerial photographs are used to update vector models. Several coordinate systems such as Universal Transverse Mercator (UTM), latitude/longitude, and local cadastral systems are used to identify locations. Data originating from different coordinate systems requires appropriate transformations. Major public sources of geographic data, including the TIGER files maintained by U.S. Department of Commerce, are used for road maps by many Web-based map drawing tools (e.g., http://maps.yahoo.com). Often there are high-accuracy, attribute-poor maps that have to be merged with low-accuracy, attribute-rich maps. This is done with a process called "rubber-banding" where the user defines a set of control points in both maps and the transformation of the low accuracy map is accomplished to line up the control points. A major integration issue is to create and maintain attribute information (such as air quality or traffic density) which can be related to and integrated with appropriate geographical information over time as both evolve.

Data Capture
The first step in developing a spatial database for cartographic modeling is to capture the two-dimensional or three-dimensional geographical information in digital form—a process that is sometimes impeded by source map characteristics such as resolution, type of projection, map scales, cartographic licensing, diversity of measurement techniques, and coordinate system differences. Spatial data can also be captured from remote sensors in satellites such as Landsat, NORA, and Advanced Very High Resolution Radiometer (AVHRR) as well as SPOT HRV (High Resolution Visible Range Instrument), which is free of interpretive bias and very accurate. For digital terrain modeling, data capture methods range from manual to fully automated. Ground surveys are the traditional approach and the most accurate, but they are very time consuming. Other techniques include photogrammetric sampling and digitizing cartographic documents.

27.4.3 Specific GIS Data Operations

GIS applications are conducted through the use of special operators such as the following:

• Interpolation: This process derives elevation data for points at which no samples have been taken. It includes computation at single points, computation for a rectangular grid or along a contour, and so forth. Most interpolation methods are based on triangulation that uses the TIN method for interpolating elevations inside the triangle based on those of its vertices.

• Interpretation: Digital terrain modeling involves the interpretation of operations on terrain data such as editing, smoothing, reducing details, and enhancing. Additional operations involve patching or zipping the borders of triangles (in TIN data), and merging, which implies combining overlapping models and resolving conflicts among attribute data. Conversions among grid models, contour models, and TIN data are involved in the interpretation of the terrain.

• Proximity analysis: Several classes of proximity analysis include computations of "zones of interest" around objects, such as the determination of a buffer around a car on a highway. Shortest path algorithms using 2D or 3D information is an important class of proximity analysis.

• Raster image processing: This process can be divided into two categories (1) map algebra, which is used to integrate geographic features on different map layers to produce new maps algebraically; and (2) digital image analysis, which deals with analysis of a digital image for features such as edge detection and object detection. Detecting roads in a satellite image of a city is an example of the latter.

• Analysis of networks: Networks occur in GIS in many contexts that must be analyzed and may be subjected to segmentations, overlays, and so on. Network overlay refers to a type of spatial join where a given network—for example, a highway network—is joined with a point database—for example, accident locations—to yield, in this case, a profile of high-accident roadways.

Other Database Functionality

The functionality of a GIS database is also subject to other considerations.
• Extensibility: GISs are required to be extensible to accommodate a variety of constantly evolving applications and corresponding data types. If a standard DBMS is used, it must allow a core set of data types with a provision for defining additional types and methods for those types.

• Data quality control: As in many other applications, quality of source data is of paramount importance for providing accurate results to queries. This problem is particularly significant in the GIS context because of the variety of data, sources, and measurement techniques involved and the absolute accuracy expected by applications users.

• Visualization: A crucial function in GIS is related to visualization—the graphical display of terrain information and the appropriate representation of application attributes to go with it. Major visualization techniques include

(1) contouring through the use of isolines, spatial units of lines or arcs of equal attribute values;

(2) hillshading, an illumination method used for qualitative relief depiction using varied light intensities for individual facets of the terrain model; and

(3) perspective displays, three-dimensional images of terrain model facets using perspective projection methods from computer graphics. These techniques impose cartographic data and other three-dimensional objects on terrain data providing animated scene renderings such as those in flight simulations and animated movies.

Such requirements clearly illustrate that standard RDBMSs or ODBMSs do not meet the special needs of GIS. It is therefore necessary to design systems that support the vector and raster representations and the spatial functionality as well as the required DBMS features. A popular GIS called ARC-INFO, which is not a DBMS but integrates RDBMS functionality in the INFO part of the system, is briefly discussed in the subsection that follows. More systems are likely to be designed in the future to work with relational or object databases that will contain some of the spatial and most of the nonspatial information.