LECTURE NOTES

ON

ADVANCED DATABASES

Course Code: AIT505

B.Tech V Sem (IARE-R16)

By

Mr. D Rahul, Assistant Professor

Mr. N Bhaswanth , Assistant Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500 043

UNIT - I

ACTIVE DATABASES

An active database system is a DBMS that supports an integrated subsystem for the definition and management of production rules (active rules). The rules follow the *event-condition-action* paradigm: each rule reacts to some events, evaluates a condition and, based on the truth value of the condition, might carry out an action. The execution of the rules happens under the control of an autonomous subsystem, known as the *rule engine,* which keeps track of the events that have occurred and schedules the rules for execution. Thus, an active database system can execute either transactions, which are explicitly initiated by the users, or rules, which are under the control of the system. We say that the resulting system exhibits a *reactive behavior* which differs from the typical passive behavior of a DBMS without active rules.

When a DBMS is active, the part of the application that is normally encoded by programs can also be expressed by means of active rules. As we shall see, active rules can, for example, manage integrity constraints, calculate derived data and manage exceptions, as well as pursue business objectives. This phenomenon adds a new dimension to the independence of the database, called *knowledge independence:* knowledge of a reactive type is removed from the application programs and coded in the form of active rules. Knowledge independence introduces an important advantage, because rules are defined with the DDL and are part of the schema, and therefore they are shared by all the applications, instead of being replicated in all the application programs. Modifications to the reactive behavior can be managed by simply changing the active rules, without the need to modify the applications.

Many prototype systems, both relational and object-oriented, provide active rules that are particularly expressive and powerful. In this chapter, we will concentrate on active databases supported by relational DBMSs; almost all relational systems support simple active rules, called *triggers,* and therefore can be considered active databases in their own right. In this chapter we will use the terms "active rule' and 'trigger' as synonymous.

Unfortunately, there is no consolidated standard proposal for triggers, as they were not defined in SQL-2. Thus, first we will give a general description, which can be adapted

easily enough to any relational system. Next, we will describe the syntax and semantics of two specific relational systems, Oracle and DB2. Covering DB2 is particularly useful because the SQL-3 standard for active rules includes a standardization of triggers that uses the same solutions as DB2. We will complete this chapter with a discussion on properties of active databases and with an illustration of their applications.

1.1 Trigger behavior in a relational system

The creation of triggers is part of the *data definition language* (DDL). Triggers can be dynamically created and dropped; in some systems they can also be dynamically activated and deactivated. Triggers are based on the event-condition-action (ECA) paradigm:

- The events are data manipulation primitives in SQL (**insert, delete, update);**
- The condition (which can sometimes be omitted) is a boolean predicate**,** expressed in SQL;
- The action is a sequence of generic SQL primitives, sometimes enriched by an integrated programming language available within the environment of a specific product (for example, PL/SQL in Oracle).

Triggers respond to events relating to a given table, called the trigger's *target.*

The ECA paradigm behaves in a simple and intuitive way: *when* the event is verified, *if* the condition is satisfied, *then* the action is carried out. It is said that a trigger is *activated* by the event, is *considered* during the verification of its condition and is *executed* if the condition is true, and therefore the action part is carried out. However, there are significant differences in the ways in that systems define the activation, consideration and execution of triggers.

Relational triggers have two levels of granularity, called *row-level* and *statement-level.* In the first case, activation takes place for each tuple involved in the operation; we say that the system has a tuple-oriented behavior. In the second case, activation takes place only once for each SQL primitive, referring to all the tuples invoked by the primitive, with a set-oriented behavior. Furthermore, triggers can have *immediate* or *deferred* functionality. The evaluation of immediate triggers normally happens immediately after the events that activated them *{after* option). Less often, the evaluation of immediate triggers logically precedes the event to which it refers (*before* option). The de-

ferred evaluation of triggers happens at the end of the transaction, following a **commit-work** command.

Triggers can activate themselves one after another. This happens when the action of a trigger is also the event of another trigger. In this case, it is said that the triggers are *cascading.* Triggers can also activate themselves one after another indefinitely, generating a computation that does not terminate. We will address this problem in Section 1.5.

1.2 Definition and use of triggers in Oracle

We will look first at the syntactic characteristics of the command to create triggers, and will then demonstrate their behavior using a typical application.

1.2.1 Trigger syntax in Oracle

The syntax for the creation of triggers in Oracle is as follows:

create trigoer *TriggerName*
Mode Event {, *Event*}
on *TargetTable*
[[referencing *Reference***]**
for each row
[when *SQLPredicate***]]**
PL/SQLBlock

The *Mode* is **before** or **after**, the *Event* is **insert**, **delete**, or **update**; **update** may be followed by attribute names of the target table. The **referencing** clause allows the introduction of variable names, for referring to the old and new values of the row that is changed, with one or both of the following clauses:

old as *OLDVanable*
| new as *NewVariable*

We will now discuss the various characteristics in detail. Each trigger controls any combination of the three DML update primitives (**insert**, **delete**. and **update**) on the target table. The granularity of triggers is determined by the optional clause for each row, which is present in the case of row-level granularity, while it is omitted in the case of statement-level granularity. The condition (*SQLPredicate*) can be present only in the triggers with row-level granularity and consists of a simple predicate on the

current tuple. Triggers with statement-level granularity, however, may substitute condition predicates with the control structures of the action part. The action, both with row and statement-level granularity, is written in PL/SQL, which extends SQL by adding the typical constructs of a programming language (as shown in Appendix *C*). The action part cannot contain DDL instructions or transactional commands.

References to the before (**old**) and after (**new**) states of the row that is modified are possible only if a trigger is row-level. In the case of insert only the after state is defined, and in the case of delete only the before state is defined. The **old** and **new** variables are implicitly available to indicate, respectively, the old and new state of a tuple. Variable names other than **old** and **new** can be introduced by the **referencing** clause.

1.2.2 Behavior of triggers in Oracle

Triggers in Oracle are immediate and allow for both the before and after options on both row- and statement-level granularity. Thus, combining the two granularities and the two functions, four combinations are obtained for each event:

before row

before statement

after row

after statement

The execution of an **insert**, **delete** or **update** statement in SQL is interwoven with the execution of the triggers that are activated by them, according to the following algorithm:

1. The before statement-level are considered and possibly executed
2. For each tuple of the target table involved in statement:
   (a) the before row-level triggers are considered and possibly executed
   (b) the statement is applied to the tuple, and then the integrity checks relative to the tuple are carried out.
   (c) the after row-level triggers are considered and possibly executed.
3. The integrity checks for the entire table are carried out.
4. The after statement-level triggers are considered and possibly executed.

If an error occurs during the evaluation or one trigger, then all the modifications carried out as a consequence of the SQL primitive that activates the trigger execution are undone. Oracle thus guarantees a partial rollback of the primitive and of all the actions caused by the triggers. Early versions of Oracle imposed a limit of one trigger per kind (before/after row/statement); recent versions have abolished these limitations, without, however, indicating how to prioritize triggers of the same kind that are activated by the same event.

The actions carried out by the triggers can cause the activation of other triggers. In this case, the execution of the current trigger is suspended and the other activated triggers are considered, by recursively applying the algorithm illustrated above. The highest number of triggers in *cascade* (that is, activated in sequence according to this schema) is 32. Once this level is reached, the system assumes that an infinite execution has occurred and suspends the execution, raising a specific exception.

1.2.3 Example of execution

We illustrate triggers in Oracle by showing them at work on a classical warehouse management problem. The **Reorder** trigger, illustrated below, is used to generate a new order automatically, by entering a tuple in the PENDINGORDERS table, whenever the available quantity, **QtyAvbl**, of a particular part of the WAREHOUSE table fills below a specific reorder level (**QtyLimit**):

```
create trigger Reorder
after update of QtyAvbl on Warehouse
when (new.QtyAvbl < new.QtyLimit)
for each row
    declare
        X number;
    begin
        select count(•) into X
        from PendingOrders
        where Part = new.Part;
        if Z = 0
        then
```

```
            insert into PendingOrdera
            values ( newPart, new.QtyReord, sysdate) ;
        end if;
    end;
```

This trigger has a row-level granularity and is considered immediately after each modification of the attribute **QtyAvbl**. The condition is evaluated row by row comparing the values of the attributes **QtyAvbl** and **QtyLimit**; it is true if the available quantity falls below the limit. The action is a program written in PL/SQL. In the program, a numeric variable **X** is initially declared; it stores the number of orders already placed for the part being considered. We assume that PENDINGOROERS is emptied when the corresponding parts are delivered to the warehouse; at each time, only one order should be present for each part. Thus, if **X** is not zero, no new order is issued. If instead **X** is zero, an order is generated by inserting a tuple into the table PENDlNGORDERS. The order contains the part numbers, the reorder quantity **QtyReord** (assumed to be fixed) and the current date. The values of the tuples that refer to the execution of the trigger are accessed by use of the correlation variable **new**. Assume that the initial content of the WAREHOUSE table is as shown in Figure 1.1.

Part	QtyAvbl	QtyLimit	Qty-Reord
1	200	150	100
2	780	500	200
3	450	400	120

Figure 1.1 initial state of the WAREHOUSE table.

Consider then the following transaction activated on 10/10/1999:

```
T1: update Warehouse
    set QtyAvbl = QtyAvbl - 70
    where Part = 1
```

This transaction causes the activation, consideration and execution of the Reorder trigger, causing the insertion into the PENDINGORDERS table of the tuple (1, 100, 10/10/1999). Suppose that next **ihe following transaction is carried** out:

```
T2: update Warehouse
    set QtyAvbl = QtyAvbl - 60
    where Part <= 3
```

The trigger is thus considered for all parts, and the condition is verified for parts 1 and 3. However, the action on part 1 has no effect, because we assume that PENDlNGORDERS still contains the tuple relating to part 1. Thus, the execution of the trigger causes the insertion into PENDINGORDERS of the single tuple (3, 120, 10/10/1999), relating to part 3.

1.3 Definition and use of triggers in DB2

In this section we will first look at the syntactic characteristics of the create trigger command, and will then discuss its behavior and an application example.

1.3.1 Trigger syntax in DB2

Each trigger in DB2 is activated by a single event, which can be any data modification primitive in SQL. As in Oracle, triggers are activated immediately, before or after the event to which they refer, and have both row and statement-level granularity. The syntax of the creation instruction for triggers is as follows:

create trigger *TriggerName*
Mode Event **on** *TargetTable*
[referencing *Reference***]**
for each *Level*
[when (*SQLPredicate***)]**
SQLProceduralStatement

where the *Mode* is **before** or **after**, the *Event* is **insert**, **delete**, or **update** (**update** may be followed by attributes of the target table), and the *Level* is **row** or **statement**. The **referencing** clause allows the introduction of variable names. If the level is row, the variables refer to the tuple that is changed; they are defined by the clauses:

old as *OldtupleVar*
| new as *NewTupleVar*

If the level is **statement**, then the variables refer to the table that is changed, with the clauses:

old_table as *OldTableVar*
| new_table as *NewTableVar*

As in Oracle, variables **old**, **new**, **old_table** and **new_table** are implicitly available, while the referencing clause enables the introduction of different variables. In the case

of insertion, only the new or new_table variables are defined; in the case of deletion, only the old and old_table variables are defined.

1.3.2 Behavior of triggers in DB2

In DB2, triggers activated before an event, the *before-triggers*, are used only to determine errors and to modify the values assigned to the new variables. These cannot contain DML commands that cause a modification of the state of the database, and thus cannot activate other triggers. The system guarantees a behavior in which the side-effects of the before-triggers become visible before the execution of the SQL primitive that activates them. The before-triggers can thus require the prior evaluation of the new values produced by the SQL primitive, which are stored in temporary data structures.

Various triggers on different levels of granularity can refer to the same event. These are considered in an order managed by the system, which takes into account their lime of creation. Row-level and statement-level triggers can be ordered arbitrarily (while in Oracle the relative ordering between triggers of different granularity is fixed, as illustrated by the algorithm in Section 1.2.2). If an action of a trigger with row-level granularity contains many SQL primitives, they are all carried out for one tuple before moving on to the next.

DB2 manuals describe precisely how the evaluation of triggers is carried out with reference to integrity constraints, in particular the referential ones, which are associated with a compensation action. Following a primitive *S*, the consideration and execution of the before-triggers are first carried out, and can cause modifications to the new variables. Then, the actions that are required for referential integrity are carried out. These actions can cause the activation of many triggers, which are added to the after-trigger activated by *S*. Finally, the system considers and executes all the activated triggers, based on their system-defined priorities. When the execution of these triggers contains SQL statements that may cause the activation of other triggers, the state of execution of the rule scheduling algorithm is saved and the system reacts by considering the triggers that were subsequently activated, thus initiating a recursive evaluation. At the end, the state of execution of the rule scheduling algorithm is restored, and the execution of the trigger that was suspended is resumed

1.3.3 Example of execution

Consider a database containing the tables PART, DISTRIBUTOR, and AUDIT. The PART table has as its primary key the attribute, **PartNum**; it has also three other attributes, **Supplier**, **City** and **Cost**. A referential integrity constraint is present in the table PART and refers to the DISTRIBUTOR table:

foreign key (Suppler)

references Distributor on delete null

Let us consider the following triggers:

- **SoleSupplier** is a before-trigger that prevents the modification of the **Supplier** attribute unless it is changed to the null value. In all the other cases, this gives an exception that forces a rollback of the primitive.
- **AuditPart** is an after-trigger that records in the AUDIT table the number of tuples modified in the PART table.

```
create trigger SoleSupplier
before update of Supplier on Part
referencing new as N
for each row
when (N.Supplier is not null)
        signal sqlstate '70005' ('Cannot change supplier')
create trigger AuditPart
after update on Part
referencing old_table as OT
for each statement
        insert into Audit
        values(user, current-date, (select count(*) from OT))
```

For example, the removal from the DISTRIBUTOR table of all the suppliers located in Penang causes the violation of the referential integrity constraint. At this point, the management policy for violation of the referential integrity constraint causes the modification to the null value of all the tuples of the PART table that remain dangling after the deletion. This activates the two triggers **SoleSupplier** and **AuditPart**. The first is a before-trigger, which is thus considered first. Its evaluation, tuple by tuple, happens

logically before the modification, but it has available the N value, which describes the variation. Thus, this value is found to be NULL, and the condition is found to be false. Finally, the **AuditPart** trigger is activated, inserting into the table AUDIT a single tuple containing the user code, the current data and the number of modified tuples.

1.4 Advanced features of active rules

Building on the basic characteristics of relational triggers, seen above, some advanced systems and prototypes of active database systems have various characteristics that increase the expressive power of active rules. Their advanced features are as follows.

Temporal and user-defined events With regard to events, these can include *temporal* or *user-defined* events. The first ones allow the expression of time-dependent events such as, for example, 'every Friday evening' or 'at 17:30 on 20/6/1999'. User defined events are explicitly named and then raised by users' programs. For instance, a 'high-water' user-defined event could be defined and then raised by an application; the raising would activate a rule that reacts to the event.

Event expressions The activation of triggers can depend not only on a single event, but also on a set of events with a simple disjunctive interpretation. Activation can also depend on generic *boolean expression of events*, constructed according to more complex operators, such as precedence among events and the conjunction of events.

Instead-of mode As well as the before and after modes, there is also another mode, called **instead of**. When the condition of the corresponding rule is true, the action is carried out in place of the activation event. However, rules with instead of modes may give rather complex and unintuitive semantics (such as: 'when updating the salary of X, instead update the salary of Y'); therefore, this clause is not present in most systems.

Detached consideration and execution The consideration and/or execution of rules can be *detached*. In this case, the consideration or execution would take place in the context of another transaction, which can be completely independent or can be co-ordinated with the transaction in which the event is verified, using mechanisms of reciprocal dependence.

Priorities The conflicts between rules activated by the same event can be resolved by *explicit priorities*, defined directly by the user when the rule is created. They can be

expressed either as a partial ordering (using precedence relations between rules), or as a total ordering (using numeric priorities). The explicit priorities substitute priority mechanisms implicitly present in the systems.

Rule sets Rules can be organized in sets and each rule set can be separately *activated* and *deactivated.*

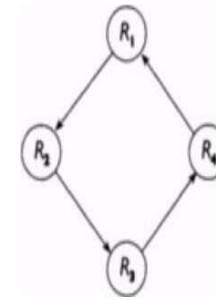
1.5 Properties of active rules

It is not difficult to design each individual active rule, once its event, condition and action are clearly identified. However, understanding the collective behavior of active rules is more complex, because their interaction is often subtle. For this reason, the main problem in the design of active databases lies in understanding the behavior of complex sets of rules. The main properties of these rules are termination, confluence and identical observable behavior.

- A set of rules guarantees *termination* when, for each transaction that may activate the execution of rules, this execution produces a final state in a finite number of steps.
- A set of rules guarantees *confluence* when, for each transaction that may activate the execution of rules, the execution terminates producing a unique final state, which does not depend on the order of execution of rules.
- A set of rules guarantees an *identical observable behavior* when for each transaction that may activate the execution of rules, this execution is confluent and all the visible actions carried out by the rule are identical and produced in the same order.

These properties are not of equal importance or desirability. In particular, termination is an essential property; we must avoid a situation in which transactions, activated by the user, cause infinite executions normally revealed by the raising of an exception when the maximum number of recursively executed rules is exceeded. Note that infinite executions are due to rules written by the database administrator, and the user would have great difficulty in understanding the situation and finding a remedy. On the other hand, confluence and identical observable behavior might not be essential, especially in the presence of various equally acceptable solutions of the same problem.

The process of *rule analysis* allows the verification of whether the properties requested are valid for a particular set of rules. In particular, an essential tool for verifying the termination of a set of rules is the *activation graph*, which represents interactions among rules. The graph is created by adding a node for each rule and an arc from a rule R_1 to a rule R_2 when the action of R_1 contains a DML primitive that is also one of the events of R_2. *A* necessary condition for non-termination is the presence of cycles in the activation graph: only in this case we can have an infinite sequence of execution of rules. An example of an activation graph is shown in Figure 1.2.



Cyclic activation graph.

Systems with many active rules are often cyclic. However, only a few cycles actually correspond to critical situations. In fact, cyclicity is a necessary but not sufficient condition for non-termination. Most cycles are indeed 'innocuous', as they describe an acceptable mutual interaction between rules.

Let us consider, for example, the rule SalaryControl (written in DB2), which creates a 'conservative' policy of salary control. It reduces the salary of all the employees when the average salary goes beyond a certain level:

```
create trigger SalaryControl
after update of Salary on Employee
then update Employee
        set Salary = 0.9 • Salary
        where (select avg(Salary) from Employee) > 100
```

The activation graph for this rule has only one node and a ring; thus, it presents a cycle, which indicates the possibility that the rule is re-activated by itself. On the other hand, whatever the initial transaction, the execution of the rule eventually terminates, as the rule progressively reduces the salaries until they are again within the established level. At this point, the condition is false. However, a slightly different rule, which gives a rise of salary rather than decreasing it, presents termination problems:

```
create trigger SalaryControl2
after update of Salary on Employee
```

tben update Employee

set Salary = 1.1 • Salary

where (select avg(Salary) from Employee) > 100

The activation graph associated with this rule does not change. However, if the rule is executed once, it will be executed an infinite number of times, causing non-termination, as the operation carried out by the rule is unable to make its condition false.

This example shows that the cycles give only 'indications' of possible causes of non-termination. A detailed analysis of cycles, which can be partly automated, can give rise to the conclusion that a cycle is innocuous, or instead to suggest modifications to rules that guarantee its termination.

1.6 Applications of active databases

Active rules respond to several application needs. Many classic applications of active rules are *internal* to the database: the active rule manager works as a subsystem of the DBMS to implement some of its functions. In this case, triggers are generated by the system and are thus not visible to the users. The typical characteristic of internal applications is the possibility of giving a *declarative specification* of the functions, from which to derive the active rules The main functions that can be entrusted to active rules of an internal type include the management of integrity constraints of a predefined structure, the calculation of derived data and the management of replicated data. Other functions include version management, privacy management, data security enforcement and event logging.

Other rules, classified as *external*, express knowledge specific to the application, which are beyond predefined and rigid schemas. These rules are also called *business rules* as they express the strategies of a company for carrying out its primary functions (see also Chapter 5 and Chapter 6). In the case of business rules, there are no fixed techniques for the derivation of rules based on specifications. Consequently, each problem must be confronted separately. Below, we will look briefly at referential integrity and then we show some business rules.

1.6.1 Referential integrity management

The management of integrity constraints using active rules requires first that the constraint be expressed in the form of an SQL predicate. The predicate will correspond to the *condition* part of one or more active rules associated with the constraint; note, however, that the predicate must be negated in the rule, so that the consideration yields a truth value when the constraint is actually violated. After this, the designer will concentrate on the events that can cause a violation of the constraint. They contribute to the *event* parts of active rules. Finally, the designer will have to decide which action to carry out following the violation of the constraint. For example, the action could be to force the partial rollback of the primitive that has caused the violation, or could carry out a *repair action*, which corrects the violation of the constraint. This is how the action part of the active rule is constructed.

We illustrate this general approach to integrity maintenance with active rules by means of the classical referential integrity constraint. Note, however, that most systems manage referential integrity by means of ad hoc methods.

We look again at the simple referential integrity constraint discussed in Section 4.1.7. Given the tables EMPLOYEE and DEPARTMENT, the constraint indicates that the **Dept** attribute of EMPLOYEE is a *foreign key* referencing the attribute **DeptName** of DEPARTMENT. The referential integrity specification is given by means of the following clause, inserted into the definition of the EMPLOYEE table:

foreign key(Dept) references Department(DeptName)
on delete set null.
on update cascade

We may consider the *foreign key* clause as a declarative specification of both the condition of the constraint and of the repair actions that must be performed to restore the database consistency. The operations that can violate this constraint are:

- **insert** into EMPLOYEE;
- **delete** from DEPARTMENT;
- **update** to EMPLOYEE**.Dept;**
- **update** to DEPARTMENT**.DeptName**.

The constraint can be expressed as an assertion for the table EMPLOYEE, which imposes for each employee the existence of a department to which the employee belongs:

```
exists (select * from Department
        where DeptName = Employee.Dept)
```

Note that this assertion indicates a property that must be true for all employees, but in an active rule. we are interested in capturing the situations that violate the constraint. We will therefore use the negation of the assertion illustrated above as the basis for building the condition to be included within the active rules:

```
not exists (select * from Department
            where DeptName = Employee.Dept)
```

The constraint can also be expressed as an assertion, already presented in negative form, for the table, DEPARTMENT. In this case, the constraint is violated if there is an employee without a department:

```
exists (select * from Employee
        where Dept not in
              (select Deptname from Department))
```

We then need to construct four active rules. Two react to each insertion in EMPLOYEE or modification of the **Dept** attribute, canceling the effect of the operations if they violate the constraint. Remember that, according to the definition of referential integrity, violations caused by operations on the internal table have to cause a rejection of the operation. The other two rules react to each deletion from DEPARTMENT or update of the **Dept** attribute, and implement the policies specified with the constraint.

The first rule is coded by the following trigger in DB2:

```
create trigger DeptRef1
after insert on Employee
for each row
when (not exists
        (select * from Department
         where DeptName = New.Dept))
signal sqlstate '70006' ('employee without department'
```

The second rule is the same except for the event:

```
create trigger DeptRef1
after update on Employee
for each row
when (not exists
        (select * from Department
         where DeptName = New.Dept))
signal sqlstate '70006' ('employee without department'
```

The third rule reacts to the cancellation of a tuple of DEPARTMENT, imposing a null value on the attribute **Dept** of the tuples involved:

```
create trigger DeptRef3
after delete on Department
for each row
when (exists
        (select * from Employee
        where Dept = Old.DeptName))
update Employee
        set Dept = null
        where Dept =Old.Deptname
```

Note that the condition is simpler than that shown above. It identifies as critical those employees whose departments coincide with a department removed by the **delete** operation. In fact, the condition could even be omitted, as the action is performed on all and only the tuples that satisfy the condition.

The fourth rule reacts to modification of the attribute **DeptName** of DEPARTMENT, reproducing on EMPLOYEE the same modification on the **Dept** attribute as in the DEPARTMENT table:

```
create trigger DeptRef4
after update of Department on Deptname
for each row
when (exists
        (select • from Employee
```

where DeptName = Old.DeptName))

update Employee

set Dept = New.Deptname

where Dept = Old.Deptname

Note that in this case, too, the condition is optimized and could even be omitted.

1.6.2 Business rules

Business rules express the strategies of a company in pursuing its objectives. Examples are the rules that describe the buying and selling of stocks based on the fluctuations in the market, rules for the management of a transport network or of energy, or rules for the management of a warehouse based on the variations of available quantities of each part (see Section 1.2.3). Some of these rules are simple *alerters,* which limit themselves to the action part and emit messages and warnings, leaving the users to manage abnormal situations.

Business rules have already been introduced in Section 5.3.1 to express schema constraints. Remember that these were classified as integrity or derivation rules. Integrity rules are predicates that express conditions that must be true. In commercial DBMSs. they can be programmed using the check clause or using *assertions.* However, many DBMSs introduce restrictions on the predicate that are expressible using these clauses, thus limiting their effective usability. Furthermore, the use of SQL-2 constraints goes together with adopting the reaction policies present in the standard (or supported by DBMSs), while the desired reaction is often different. Therefore, active rules (which are supported by most relational DBMSs) can be used for the specification and implementation of 'generic' constraints and 'arbitrary' reactions.

Let us look at how we can program the business rule BR2 introduced in Section 5.3.1, using an active rule. The business rule is repeated here: (BR2) *an employee must not have a salary greater than that of the manager of the department to which he or she belongs.*

Let us use the tables EMPLOYEE and DEPARTMENT, where **EmpNum** is the primary key of EMPLOYEE and **DeptNum** is the primary key of DEPARTMENT; EMPLOYEE has the attributes **MgrSalary**, and **DeptNum** and DEPARTMENT has the attribute **Director**. The operations that can violate the constraint are the update of

the salary of the employees in their double role as employee and manager, and the insertion of a new employee Let us suppose that among these, the critical modification to be monitored is the increase in the salary awarded to an employee. Let us also suppose that the reaction policy is to block the update, and to signal this behaviour. These choices correspond to the following trigger, written using the DB2 syntax:

```
create trigger ExcessiveSalary
after update on Salary of Employee
for each row
when New.Salary > select Salary
                  from Employee
                  where EmpNum in
                        (select Director
                        from Department
                        where DeptNum = New.DeptNum)
then signal sqlstate '70005' ('Salary too high')
```

The rules concerning warehouse management or the handling of suppliers illustrated in Section 1.2.3 and Section 1.3.3, can be considered as other examples of application-specific business rules.

Business rules are particularly advantageous when they express the reactive policies at schema level (and are thus valid for all applications) because they allow an unambiguous and centralized specification. This allows the property of *knowledge independence,* discussed in the introductory section to this chapter.

1.7 Properties of Active Rule Execution

Designing individual active rules is not too difficult, once it is well understood that the rule reacts to a give n event, tests a given condition, and performs a given action. However, understanding the collective behavior of active rules is much more difficult than just observing them individually because rule interactions are often subtle and unexpected. Thus, the main problem in the design of an active rule set regards their collective behavior. Termination, confluence, and observable determinism are the most relevant properties for understanding the collective behavior of a set of active rules:

- A rule set guarantees termination when, for any user-defined transaction triggering the processing of rules, the processing eventually terminates, producing a final state.
- A rule set guarantees confluence when, for any user-defined transaction triggering the processing of rules, the processing eventually terminates, producing a unique final state that does not depend on the order of execution of the rules.
- A rule set guarantees observable determinism when, in addition to confluence, for any user-defined transaction, all visible actions performed by rules (including alerting by means of messages or output production) are the same.

These abstract properties are not equally important or desirable; in the following, we consider each of them separately. The process of checking, at rule design time, that the above properties hold is termed rule analysis. Rule analysis is performed through both formal, automatic techniques and informal reasoning.

1.8 Rule Modularization

Modularization is a key design principle in software design. It enables the designer to focus on subsets of the original problem, thus partitioning a large design space; in software engineering, modularization enables the separation of programming "in the small" from programming "in the large." With active rules, modules (or groups) typically consist of subsets of the entire active rule set, which are put together due to shared properties (e.g., all rules dealing with a given applicative problem, or defined by a given user, or relative to a given target). In this section, we discuss modularization relative to termination, which is the main design problem of active rules.

The modularization approaches of active rules for providing termina-tion are called stratifications. The term "stratification" was introduced in deductive databases to denote the partitioning of deductive rules into com-ponents, so that the ordering of components indicates a precedence in the evaluation of rules; when rules are not stratified, their semantics relevant to negation or aggregates gives rise to a number of problems, discussed in Part

Stratification in active databases induces a partitioning of active rules into components, so that termination can be determined by reasoning only within components; the designer can abstract rule behavior by reasoning locally on each individual stratum separately and then reasoning globally on the behavior across strata. The ordering of components in the stratification of active rules yields a "preferred order" for their evaluation. Three approaches to stratification are possi-

ble:

- Behavioral stratification associates each stratum to a particular applicative task; each stratum is responsible for performing the task. Global termination requires that interleaved executions of rules from different strata be under control, so that the task being pursued by one stratum is not compromised by rules from other strata.
- Assertion stratification associates each rule to an assertion, called the stratum's post condition. Global termination requires that inter-leaved executions of rules from different strata do not compromise the post conditions that were already established.
- Event-based stratification defines a stratum in terms of the input/ output relationship between its triggering events and its actions. Global termination requires that input/output events of all strata have some global acclivity property.

1.9 Rule Debugging and Monitoring

Although rule analysis and modularization, performed at compile time, should drive active rule design, the run-time debugging and monitoring of active rules is sometimes required in order to tune their behavior. Commer-cial systems, however, are rather inadequate for this purpose. Often, they do not even offer a trace facility for knowing which active rules have been running, so the only monitoring that can be performed is by looking at the rules' actions, which, in turn, become known only through the inspection of the database. In contrast, several research prototypes are focused on rule debugging; in particular, we present the features of the debugger of Chimera.

The debugger can be invoked by a user by issuing a special command, or it can be started during rule processing when given rules are being considered or executed. In order to do so, it is possible to set spy points in the rules. When the debugger is called into action, the execution of the current rule is completed (if rules are being processed), after which execution proceeds interactively; execution of rules is available at two levels of granularity:

- At the rule step level: After each rule execution, rule processing is halted and the situation is presented to the user. At this level there is no possibility of interrupting the run-time system during the computation of triggered rules and the evaluation of rules' conditions.

- At the intra-rule step level: Rule processing is halted at each of the three fundamental moments of rule execution (triggering, condition evaluation, and action execution). The user can obtain information on the state of rule processing and influence (in a limited way) the behavior of the rule executor.

The following functionalities are available in both modes:

- Information on rules: The system displays all available information about rules, including their source code, triggering time, and event consumption mode.

- Commands for rule activation and deactivation: Rules can be explicitly deactivated during debugging, so that they are disregarded during subsequent rule processing. A deactivated rule can be reactivated at any time.

- Inspection of the conflict set: The system displays all rules that are currently triggered in order of priority.

- Inspection of the deferred conflict set: The system displays those trig-gered rules whose execution is deferred.

- Inspection of the trace: The system displays all rules that were considered or executed since the last quiescent point.

- Information on occurred events: The system lists all events since the beginning of the transaction. Each event is described by its event type, the list of the OlDs of the objects affected by the event, and the indication of the database state in which the event has occurred. For identifying the various states of the database, all intermediate states since the beginning of the transaction are numbered progressively. Events are listed in order of occurrence.

When the debugging mode is intra-rule step, the following additional options are available:

- Display of the processing status: A graphical icon is used to show the current point of execution, either the computation of triggered rules, the evaluation of a rule's condition, or the execution of a rule's action.

- Detection of the next executable rule: Finds the rule with highest priority, among those that have not been considered yet, whose condition is satisfied in the current database state.

- Modification of dynamic priorities: Alters the chosen priority of trig-gered rules with equal static priority. In this way, it is possible to force the selection of a different rule from the one that would be chosen using the built-in conflict resolution strategy.

- Information on bindings produced by the condition's consideration: En-ables the inspection of the objects that are bound by the evaluation of the rule's condition.

1.10 IDEA Methodology

The use of objects and rules in modern database systems is the main focus of the IDEA Esprit Project. In particular, it inspired the IDEA Methodology, a comprehensive and systematic approach to the design of database appli-cations that use both deductive and active rules. The IDEA Methodology reconciles deductive and active rules by assigning them the role of express-ing knowledge about the application domain, either with a purely declarative style or with a more procedural style.

The IDEA Methodology extends recently published object-oriented soft-ware engineering methodologies, targeted toward arbitrary software systems and typically leading to implementations supported by an object-oriented programming language, such as C++ or Smalltalk. Conversely, the IDEA Methodology focuses on information systems (e.g., software systems managing large amounts of structured data).

Objects, deductive rules, and active rules are the three ingredients of the IDEA Methodology; each of them is fundamental for a precise conceptual description of an information system. Objects provide encapsulation as a form of abstraction that enables the designer to structure its applications. Deductive and active rules can be used to establish and enforce data management policies, as they can provide a large amount of the semen-tics that normally needs to be coded with application programs; this trend in designing database applications, called knowledge independence, brings the nice consequence that data management policies can effectively evolve just by modifying rules instead of application programs. Even if objects and rules are not yet fully supported by products, nevertheless their com-binned use at the conceptual level generates a better understanding of the overall application.

Like most object-oriented methodologies, the IDEA Methodology includes the three classical phases of analysis, design, and implementation. In addition, it includes prototyping as an intermediate phase, placed be-tween design and implementation, and dedicated to verification and critical assessment of the conceptual schemas.

1. Analysis is devoted to the collection and specification of requirements at the conceptual level. This phase is focused on modeling reality with semi-formal, expressive representation devices, aiming at a natural and easy-to-understand representation of the "universe of discourse." Therefore, this phase uses conceptual models with an associated graphical representation that are well established in software engineering practice, such as the Enti-ty-Relationship model and State charts.

2. Design is the process of translating requirements into design documents that provide a precise, unambiguous specification of the application. Design is conducted by mapping from semiformal specifications into fully formal, computer-process able specifications. The process is di-vided into schema de-sign (concerned mostly with types, classes, relationships, and operations) and rule design (further subdivided into deductive rule design and active rule de-sign).

3. Rapid prototyping is the process of testing, at a conceptual level, the adequacy of design results with respect to the actual user needs. A variety of formal transformation methods can be applied to improve the quality of the design, to verify its formal properties, or to trans-form design specifications into equiva-lent specifications that exhibit different features. Tools, which are available on the Internet, assist the automatic generation and analysis of active rules, and enable the prototyping of applications written in Chimera.

4. Implementation is the process of mapping conceptual specifications into schemas, objects, and rules of existing database platforms; the process is in-fluenced by the features of the specific target environments that were selected. These include Oracle, Illustrate, and DB2, three classic relational products supporting triggers; ODE, an object-oriented database available on the Internet to universities and research institutes; and Validity, the first deductive and ob-ject-oriented database system that will be soon be brought to the market.

1.10.1 Active Rule Design

Active rule design in the IDEA Methodology considers two different ap-proaches for internal and external rules.

The design of external rules follows a declarative approach, consisting of giving a declarative specification of active rules and then semiautomat-ically generating them with generation algorithms (supported by suitable rule generation tools). The rationale

of this approach is that a generation algorithm is able to generate rules that satisfy given quality criteria, in particular guaranteeing termination and/or confluence.

Integrity constraints constitute a natural application of this approach. A constraint on a database can be represented as a condition that must always be false. From a declarative specification of constraints, a user can easily generate a set of active rules capable of guaranteeing the consistency of the database; it is sufficient to write an abort rule as defined in Section 3.1. This simple solution to the problem does not use all the power of active rules because the reaction consists simply in discarding all the work done by the transaction; thus, a tool developed for assisting the IDEA Methodology is able to automatically generate repair rules, as defined in Section 3.1, implementing repairing policies that heuristically try to maximize the user's satisfaction, expressed informally.

Maintenance of materialized views is another classical application of rule generators; several approaches have been developed for the incremental maintenance of materialized views, as defined in Section 3.2. In the IDEA Methodology we classify rules into classes and then generate active rules to maintain views according to the mapping technique that applies to each class.

The design of business rules requires understanding the business process, and in particular the applicative goal that is pursued by the rules. In order to understand this goal, it is convenient to associate rules with a metric that measures the progress in achieving the task's objective. This goal-directed design of active rules is useful both for designing individual rules and for understanding their interaction and modularization. The following overall design strategy is suggested:

1. Identify applicative tasks for active rules. Associate each task to the condition under which the task should be executed. Give a simple description of the task in the form: "if condition, then action."

2. For each task, detect the events that cause the task to be executed; for each task, identify a metric that indicates the "progress" toward the solution of the task.

3. Generate active rules responding to the events that are associated with the task. The designer should constantly check that rules, if running, improve the metric and thus "progress" toward the task's solution.

1.10.2 Active Rule Prototyping

Prototyping denotes a methodological phase in which design results are tested; to this purpose, design results are implemented on a small scale, typi-cally with rapid prototyping software, and their adequacy and conformity to requirements are evaluated by designers and by users. During prototyping we look at rule collections, regardless of the techniques that are required in order to collect them; thus, we consider a new aspect of knowledge design, called knowledge design in the large; in contrast, the design techniques for individual active and deductive rules can be regarded as knowledge design in the small. Active rule prototyping in the IDEA Methodology has two facets: com-pile-time rule analysis, which can be used in order to prove properties of active rules, and run-time testing, which can be used to experiment with rules in order to assess and fine-tune their behavior. Both analysis and testing are assisted by rapid prototyping environments made available by the IDEA Project.

1.10.3 Active Rule Implementation

During the final implementation phase, the conceptual specifications are mapped into schemas, objects, and rules of five existing database platforms. Although all the selected database platforms implement some form of active rules, the mapping of active rules is difficult and problematic because of the intrinsically operational nature of active rule semantics, which is quite different in the five proposed systems, and also because of the heavy limitations that each product is introducing with respect to active rules supported in the conceptual model. In the end, two specific mapping techniques have emerged, used primarily for the mapping to relational products.

Meta-triggering uses the native active rule engine in order to detect events, but then requires a second active engine in order to render the semantics of conceptual rules; in practice, the second active engine is capable of executing the Chimera rule processing algorithm on each of the selected target systems. In this way, meta-triggering preserves all features of conceptual active rules. Typically, the second active engine is programmed by using stored procedures and imperative language attachments; it is application-independent and therefore can be reused for developing arbitrary applications.

Macro-triggering uses instead just the native active rule engine avail-able on each target system; conceptual triggers are aggregated to con-stitute macro-triggers, defined on each target system. Macro-triggers normally do not have the same semantics as the conceptual active rules, but differences are well identified.

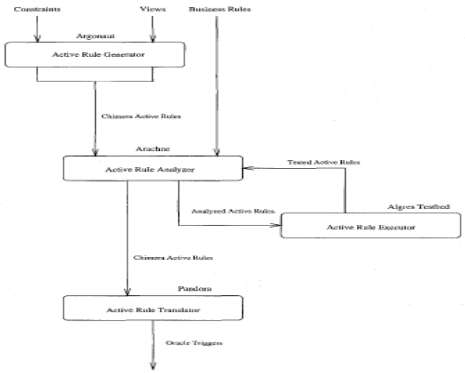
1.10.4 Design Tools Supporting the IDEA Methodology

A complete tool environment for assisting the design of active rules applications was developed at Polytechnic di Milano in the context of the IDEA Project, for supporting the IDEA Methodology. The architecture of the tools provided in the IDEA design environment is represented in Figure 4.3. lade is used during analysis in order to collect the schema specifications by means of the Object Model (an extended entity-relationship model). These specifications are semi automatically mapped into schema declarations in Chimera and into constraints and triggers useful to preserve schema integrity.

Argonaut supports the generation of active rules from the declarative specification of integrity constraints and views. In the former case, active rules correct integrity violations; in the latter case, they incrementally maintain materialized views corresponding to both non recursive and recursive deductive rules.

Arachne supports the compile-time termination analysis of a set of Chimera active rules. The tool determines the potential causes of infinite executions originating from the mutual interaction of active rules. Two differ-ent types of analysis are performed: a syntactic analysis that compares the events produced by rule actions with the events triggering the rules, and a more complex analysis that also considers rule conditions in order to detect irrelevant interactions in advance.

The Algres Tested is an execution environment that permits the rapid prototyping of the design specifications. It provides functionalities for brows-ing schemas and rules and for monitoring the execution of transactions and active rules. The run-time debugger includes several features for temporarily changing the parameters of active rules (their coupling mode, priority, and event consumption mode). If any active rule is modified during the testing session, the Arachne rule analyzer can be interactively called to analyze the new rule set, thus verifying the correctness of the performed modifications.


Constraints
Views
Business Rules
Argonaut
Active Rule Generator
Chimera Active Rules
Arachne
Active Rule Analyzer
Tested Active Rules
Algres Testbed
Analyzed Active Rules
Active Rule Executor
Chimera Active Rules
Pandora
Active Rule Translator
Oracle Triggers

UNIT – II
TEMPORAL DATABASES

Time is a pervasive aspect of reality. Events occur at specific points in time; objects and the relationships among objects exist over time. The ability to model this temporal dimension of the real world is essential to many computer applications, such as accounting, banking, inventory con-trol, econometrics, law, medical records, land and geographical information systems, and airline reservations.

A temporal *database* is one that supports some aspect of time. This part summarizes the major concepts, approaches, and implementation strategies underlying temporal databases.

This part commences with a case study that illustrates the difficulties a developer encounters when implementing via SQL an application that man-ages time-varying information. We then consider time itself, in particular how it is modeled. The design space of temporal data models is surveyed. Chapter 5 ends by listing the many temporal query languages that have been defined.

2.1 The Time Domain

Models of time in a temporal logic represent time as an arbitrary set of instants with an imposed partial order. Additional axioms introduce other, more refined models of time. For example, linear time can be specified by adding an axiom imposing a total order on this set. In the linear model, time advances from the past to the future in a step-by-step fashion. In the branching model, also termed the *possible* futures or hypothetical model, time is linear from the past to now, where it then divides into several time lines, each representing a potential sequence of events. Along any future path, additional branches may exist. The structure of branching time is a tree rooted at now. Generalizations allow branches in the past, or allow branches to join. Recurrent processes may be associated with a cyclic model of time. An example is a week, in which each day recurs every seven days.Axioms may also be added to temporal logics to characterize the density of the time line. Combined with the linear model, discrete models of time are isomorphic to the natural numbers, implying that each point in time has a single successor. Dense models of time are isomorphic to either the rationals or the reals: between any two moments of time another moment exists. Continuous models of time are isomorphic to the reals, that is, they are both dense and, unlike the rationales, contain no "gaps."In the continuous model, each real number corresponds to a "point" in time; in the discrete model, each natural number corresponds to a non-

decom-posable unit of time with an arbitrary duration. Such a nondecomposable unit of time is referred to as a chronon. A chronon is the smallest duration of time that can be represented in this model. It is not a point, but a line segment on the time line. Although time itself is perceived by most to be continuous, the discrete time model is generally used. Several practical arguments justify this choice. First, measures of time are inherently imprecise. Clocking instruments in-variably report the occurrence of events in terms of chronons, not time "points." Hence, events, even so-called instantaneous events, can at best be measured as having occurred during a chronon. Second, most natural language references to time are compatible with the discrete time model. For example, when we say that an event occurred at 4:30 PM, we usually don't mean that the event occurred at the "point" in time associated with 4:30 PM, but at some time in the chronon (perhaps minute) associated with 4:30 PM. Third, the concepts of chronon and period allow us to naturally model events that are not instantaneous but have duration. Finally, any implementation of a data model with a temporal dimension will of necessity have to have some discrete encoding for time.

Axioms can also be placed on the boundedness of time. Time can be bounded orthogonally in the past and in the future. A finite encoding implies bounds from the left (i.e., the existence of a time origin) and from the right. Models of time may include the concept of distance, though most temporal logics do not do so.

Finally, one can differentiate relative time from absolute time (more pre-cise terms are unanchored and anchored). For example, "9 AM, January 1, 1996" is an absolute time, and "9 hours" is a relative time. This distinction, though, is not as crisp as we would hope, because absolute time is with re-spect to another time (in this example, midnight, January 1, AD 1), termed an anchor. Relative time can be distinguished from distance in that the former has a direction. For example, you could envision a relative time of -9 hours, but distance is unsigned.

2.2 Time Data Types

Several temporal data types have proven useful. The most basic is a time instant, which is a particular chronon on the time line. An event is an instantaneous fact, that is, something occurring at an instant. The event occurrence time of an event is the instant at which the event occurs in the real world.

SQL-92 provides three instant data types: **DATE** (a particular day, with a year in the range AD 1-9999), TIME (a particular second within a range of 24 hours), and TIMESTAMP (a particular fraction of a second, defaulting to microsecond, of a par-

ticular day).

A time *period* is the time between two instants. In some of the liter-ature, this notion is called a time interval, but this usage conflicts with the SQL-92 data type INTERVAL, which is a different concept altogether. SQL-92 does not include periods, but periods are now part of the evolving SQL3 specification.

A time interval is a directed duration of time, that is, an amount of time with a known length, but not specific starting or ending instants. A positive interval denotes forward motion of time, toward the future. SQL-92 supports two kinds of intervals, month-year and second-day intervals.Two final temporal data types are instant *sets*, which are (logically!) sets of instants, and temporal elements, which are finite unions of periods. Temporal types must be representable. A bounded discrete represen-tation, as an integer count of the instants since the origin, is the simplest option. A bounded dense representation is also not difficult to manage, as all rationals may be expressed as the ratio between two integers. A floating point representation may also be employed. A continuous representation is the most difficult to implement.

2.3 Associating Facts with Time

The previous sections discussed the time domain itself. We now turn to associating time with facts.

2.3.1 Dimensionality

In the context of databases, two time dimensions are of general interest: the valid time dimension and the transaction time dimension.

Valid time concerns the time a fact was true in reality. The valid time of an event is the time at which the event occurred in the real world, independent

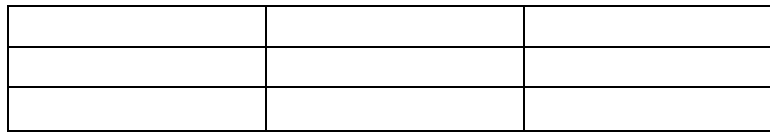


Figure 2**.1:** Structure of a snapshot relation

of the recording of that event in some database. Valid times can also be in the future, if it is expected that some fact will be true at a specified time after now.Transaction time concerns the time the fact was present in the database as stored data. The transaction time (a period) of a fact identifies the transaction that inserted the fact into the

database and the transaction that removed this fact from the database.

These two dimensions are orthogonal. A data model supporting neither is termed snapshot, as it captures only a single snapshot in time of both the database and the enterprise that the database models. A data model supporting only valid time is termed valid-time, one that supports only trans-action time is termed transaction-time, and one that supports both valid and transaction time is termed bitemporal. Temporal is a generic term implying some kind of time support.

2.3.2 Underlying Data Model

Time has been added to many data models: the entity-relationship model, semantic data models, knowledge-based data models, and deductive data models. However, by far the majority of work in temporal databases is based on the relational and object-oriented models. For this reason, we focus on these two data models in our subsequent discussion.

2.3.3 Valid Time

These models may be compared along the valid-time dimension by asking two basic questions: how is valid time represented and how are facts associated with valid time. Categorizes most of the data models along these two aspects. We do not include the OODAPLEX, Sciore-1, and TIGUKAT data models, as these two aspects are arbitrarily specifiable in these models.

Valid times can be represented with single chronon identifiers (i.e., in-stant timestamps), with periods (i.e., as period timestamps), or as *valid-time* elements, which are finite sets of periods. Valid time can be associated with papers describing these models may be found in the bibliographic notes at the end of this chapter.

2.3.4 Transaction Time

The same general issues are involved in transaction time, but there are about three times as many alternatives. The choices made in the various data models are characterized in Table 5.4. OODAPLEX is not included, as it can support virtually any of these options (while that is also possi-ble in TIGUKAT, specific support for versioning has been added to the data model and language). Transaction time may be represented with the following

2.3.5 Representative Data Models

To ground this discussion, let's examine five representative models. One of the simplest is Segev's valid-time data model, in which tuples are time-stamped with the instant that the tuple became valid. This allows the history of the attribute values of a key to be succinctly captured. In the following relation instance, we see that Eric started working in the shoe department on June 1 (in these examples, we omit the month and year from the timestamp). He moved to the book department on June **6,** and returned to the shoe department on June 11. He resigned on June 13; this requires a separate tuple, with null values for all the nonkey attributes.

Name	*Dept*	*Time*
Eric	Shoe	1
Eric	Book	6
Eric	Shoe	11
Eric	Null	13

This data model can use such a simple timestamp because it does not permit multiple values at any point in time. By using period timestamps, as for example in Sarda's data model, multiple values can be accommo-dated. The following shows the same information as above, in a period-timestamped model.

Name	*Dept*	*Time*
Eric	Shoe	[1-5]
Eric	Book	[6-10]
Eric	Shoe	[11-13]

Note that null values are not required in Sarda's model when an em-ployee resigns.

Several of the models timestamp attribute values instead of tuples. This allows more history to be captured in a single tuple. In the HRDM, attribute values are functions from time to a value domain:

2.4 Temporal Query Languages

A data model consists of a set of objects with a specified structure, a set of constraints on those objects, and a set of operations on those objects. In the two previous sections we have investigated in detail the structure of, and constraints on, the objects of tem-

poral databases. Here, we complete the picture by discussing the operations, specifically temporal query languages. Many temporal query languages have been proposed. In fact, it seems that each researcher feels it necessary to define a new data model and query language. Lists the major temporal query language proposals to date. The underlying data model is a reference. The next column lists the conventional query language the temporal proposal is based on. Most of the query languages have a formal definition.

Lists the object-oriented query languages that support time. Note that many "nested" relational query languages and data models, such as HQuel, HRDM, HTQuel, TempSQL, and TBE, have features that might be considered to be object-oriented.

The data model and conventional query language on which the temporal query language is based are identified in the second and third columns. The fourth column indicates whether the language has been implemented. It is rare for an object-oriented query language to have a formal semantics. Also in contrast to temporal relational query languages, most object-oriented query languages have been implemented.

2.5 TSQL2

The Temporal Structured Query Language, or TSQL2, was designed by a committee of 18 researchers who had individually designed many of the lan-guages listed in the previous chapter. The goal of TSQL2 was to consolidate approaches to temporal data models and calculus-based query languages, to achieve a consensus extension to SQL-92 and an associated data model upon which future research could be based. Additionally, TSQL2 is being incorporated into the evolving SQL3 standard.

2.5.1 Time Ontology

TSQL2 uses a linear time structure, bounded on both ends. The origin is 18 billion years ago, when the Big Bang is thought to have occurred; the time line extends 18 billion years into the future.

The TSQL2 time line is a discrete representation of the real time line, which can be considered to be discrete, dense, or continuous. The TSQL2 time line consists of atomic (nondecomposable) chronons. Consecutive chronons may be grouped together into granules, with different groupings yielding distinct granularities. TSQL2 allows a value of a temporal data type to be converted from one granularity to another.

TSQL2 is carefully designed not to require choosing among the discrete, dense, and continuous time ontologies. Rather, TSQL2 permits no question to be asked that would differentiate among these three ontologies. For ex-ample, it is not possible to

ask if an instant a precedes an instant b. It is only possible to ask that question in terms of a specified granularity, such as seconds, days, or years. Different granularities could yield different answers to this question. Similarly, distance is in terms of a specified granularity, and is represented as an integral number of granules.

TSQL2 inherits the temporal types in SQL-92, **DATE,** TIME, TIME-STAMP, and INTERVAL, and adds the PERIOD data type.

2.5.1 Data Model

TSQL2 employs a very simple underlying data model. This data model retains the simplicity and generality of the relational model. It has no illu-sions of being suitable for presentation, storage, or query evaluation. Instead, separate, representational data models, of equivalent expressive power, are employed for implementation and for ensuring high performance. Other presentational data models may be used to render time-varying behavior to the user or application. A coordinated suite of data models can achieve in concert goals that no single data model could attain. Employee Jake was hired by the company as temporary help in the shipping department for the period from time 10 to time 15, and this fact became current in the database at time 5. This is shown in Figure 6. 1(a). The arrows pointing to the right signify that the tuple has not been logically deleted; it continues through to the transaction time until changed *(U.* C.).

2.5.2 Language Constructs

We now turn to the statements available in TSQL2.

2.5.2.1 Schema Definition

This language is a strict superset of SQL-92, and so it supports conventional relations in all their grandeur. To explore the temporal features of TSQL2, we'll need a tempor-al relation. Envision a patient database at a doctor's office. Included in this database is information on the drugs prescribed to each patient.

Example : Define the Prescription relation

```
CREATE TABLE Prescription (Name    CHAR(30),

                Physician  CHAR(30),Drug CHAR(30), Dosage CHAR(30),
                Frequency INTERVAL  MINUTE)
        AS VALID  STATE DAY  AND  TRANSACTION
```

The Name column specifies the patient's name. The Frequency is the number of mi-nutes between drug administrations.

The **AS** clause is new in TSQL2. The valid time specifies the period(s) during which the drug was prescribed. The transaction time specifies when this information was recorded as current in the database. Tuples that have not been updated or deleted will have a transaction time that includes now.

The valid time has a granularity of 1 day. The granularity of the trans-action time is system-dependent, but most likely will be no coarser than a millisecond, to differentiate consecutive transactions.

The Prescription relation is a bitemporal state relation, as it includes both kinds of time. There are six kinds of relations:

- snapshot relations, which have no temporal support
- valid-time state relations, specified with **AS** VALID **STATE (STATE** is optional)
- valid-time event relations, specified with **AS** VALID **EVENT** "
- Transaction-time relations, specified with **AS TRANSACTION**
- Bitemporal state relations, specified with **AS** VALID **STATE AND TRANSACTION**
- Bitemporal event relations, specified with **AS** VALID **EVENT AND TRANSACTION**
- The type of a relation can be changed at any time, using the ALTER TABLE statement.

Example : What drugs have been prescribed with Proventil?

```
SELECT Pl.Name, P2.Drug
FROM Prescription AS P1, Prescription AS P2
WHERE P1.Drug = 'Proventil' AND P2.Drug <> 'Proventil'
        AND P1.Name = P2.Name
```

The result is a set of tuples, each specifying a patient and a drug, along with the maximal period(s) during which both that drug and Proventil were prescribed to that patient.

Restructuring

One of the most powerful constructs of TSQL2 is restructuring. Whereas TSQL2 automatically performs coalescing on the result of a query, restruc-turing in the FROM clause allows coalescing to be performed on the under-lying tuples.

Example: Who has been on a drug for more than a total of six months?

```
SELECT Name, Drug
FROM Prescription(Name, Drug) AS P
WHERE CAST(VALID(P) AS INTERVAL MONTH)
        > INTERVAL '6' MONTH
```

Notice that the FROM clause mentions in parentheses several of the at-tributes of the Prescription relation. This clause projects out the Name and Drug attributes, then coalesces the result, which is then manipulated in the remainder of the query. By restructuring on Name and Drug, the timestamp associated with each name-drug pair indicates the maximal period(s) when that patient was prescribed that drug, independent of the prescribing physi-cian, the dosage, or the frequency. Hence, a single pair may be computed from many pairs of the underlying Prescription relation. The other attributes are not available via P.

The new VALID(P) construct returns the valid-time element (set of max-imal periods) associated with P. Then, the **CAST** operator converts it to the type INTERVAL **MONTH** by summing the durations (in months) of each of the maximal periods. This computes the total number of months that patient has been prescribed that drug, ignoring gaps when the drug was not prescribed. This total is compared with the interval constant **6** months.

The result is a relation with two columns, the patient's name and the drug, along with a timestamp specifying when that drug was prescribed.

2.6 Adding Temporal Support

In the following, we visit each of these components in turn, reviewing what changes need to be made to add temporal support.

DDL Compiler

The changes to support time involve adding temporal domains, such as pe-riods, and

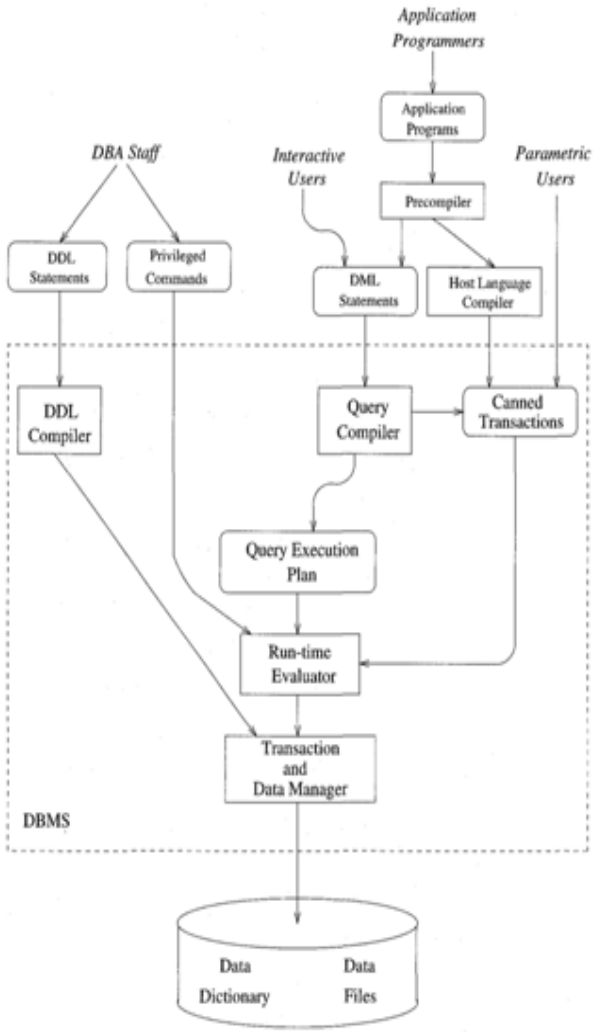
adding constructs to specify support for transaction and valid time in the definition of relations. A more substantial change is the data dictionary, which must now consist of transaction-time relations. Schema versioning concerns only the recording of the data, and hence does not in-volve valid time. The attributes and their domains, the indexes, and even the names of the relations vary over transaction time.

Query Compiler

Optimization of temporal queries is more involved than that of conventional queries, for several reasons. First, optimization of temporal queries is more critical, and thus easier to justify expending effort on, than conventional optimization. The relations that temporal queries are defined over are larger, and are growing monotonically, with the result that unoptimized queries take longer and longer to execute. This justifies trying harder to optimize the queries, and spending more execution time to perform the optimization.

Second, the predicates used in temporal queries are harder to optimize. In traditional database applications, predicates are usually equality predi-cates (hence the prevalence of equijoins and natural joins); if a less-than join is involved, it is rarely in combination with other less-than predicates. On the other hand, in temporal queries, less-than joins appear more frequently, as a conjunction of several inequality predicates. As an example, the TSQL2 OVERLAPS operator is translated into two "<" predicates on the underly-ing timestamps (see Example 5.3). Optimization techniques in conventional databases focus on equality predicates, and often implement inequality joins as Cartesian products, with their associated inefficiency.

However, there is greater opportunity for query optimization when time is present. Time advances in one direction; the time domain is continuously expanding, and the most recent time point is the largest value in the domain. This implies that a natural clustering or sort order will manifest itself, which can be exploited during query optimization and evaluation. The integrity constraint **BEGIN(p) PRECEDES END(p)** holds for every period p. Also, for many relations, the periods associated with a key are contiguous in time,



Components of a database management system

with one period starting exactly when the previous period ended. An ex-ample is salary data, where the periods associated with the salaries for each employee are contiguous. Semantic query optimization can exploit these integrity constraints, as well as additional ones that can be inferred.

The importance of efficient query optimization and evaluation for tem-poral databases was underscored by an initial study that analyzed the per-formance of a brute-force approach to adding time support to a conventional DBMS. In this study, the university Ingres DBMS was extended in a min-imal fashion to support TQuel. The results were discouraging. Sequential scans, as well as access methods such as hashing and ISAM, suffered from rapid performance degradation due to ever-growing overflow chains. Because adding time creates multiple tuple versions with the same key, reorganization did not help to shorten overflow chains. The objective of work in temporal query evaluation, then, is to avoid looking at all of the data, because the alternative implies that queries will continue to slow down as the database accumulates facts. We emphasize that these results do not imply that a time-varying database implemented on a conventional DBMS will be much less efficient than that database implemented on a brute-force temporal DBMS. In fact, simulating a time-varying database on a conventional DBMS, which is currently the only alternative available to application programmers, will encounter all of the problems listed above.

A single query can be optimized by replacing the algebraic expression with an equivalent one that is more efficient, by changing an access method associated with a particular operator, or by adopting a particular implemen-tation of an operator. The first alternative requires a definition of equivalence in the form of a set of tautologies. Tautologies have been identified for many of the algebras listed in Table 5.5. Some of these temporal algebras support the tautologies defined in the standard relational algebra, enabling existing query optimizers to be used.

To determine which access method is best for each algebraic operator, metadata, that is, statistics on the stored temporal data, and *cost mod*els, that is, predictors of the execution cost for each operator implementation/access method combination are needed. Temporal data requires additional metadata, such as the time period over which the relation is defined (termed the lifespan), the lifespans of the tuples, the surrogate and tuple arrival distributions, the distributions of the time-varying attributes, regu-larity and granularity of temporal data, and the frequency of the null values that are sometimes introduced when attributes within a tuple aren't synchronized. Such statistical data may be updated by random sampling or by a scan through the entire relation. In particular, selectivity estimates on the size of the results of various temporal joins have been derived.

Run-Time Evaluator

A wide variety of binary joins have been considered, including time-join and time-equijoin (TE-join), event-join and TE-outerjoin, contain-join, contain-semijoin and intersect-join, and temporal natural join. The various algo-rithms proposed for these joins have generally been extensions to nested loop or merge joins that exploit sort orders or local workspace, as well as hash joins.

Several approaches have been proposed for implementing temporal joins. Some of these exploit ordering of the input tables to achieve higher efficiency. If the underlying tables are ordered, coalescing can be handled in a manner similar to that for projection.

Coalescing is an important operation, since value-equivalent tuples may be present in the representation. Also, the semantics of some queries demand that the input relations be coalesced prior to evaluation. If prior coalescing is required, this is most easily accomplished if the input relation is sorted on the explicit attribute values. The temporal element associated with the conceptual tuple is easily reconstructed during a scan of the relation. If indexes or precomputed results are available, then it may be possible to avoid the relation scan.

We note that for many predicates prior coalescing is not required. For example, if a preicate references only the explicit attributes of a relation, then the coalescing operation can be eliminated.

Conventional indexes have long been used to reduce the need to scan an entire relation to access a subset of its tuples. Indexes are even more important in temporal relations that grow monotonically in size. Many temporal indexing strategies are available. Most of the indexes are based on B+-trees, which index on values of a single key; the remainder are based on R-trees, which index on ranges (periods) of multiple keys. The worst-case performance for most proposals has been evaluated in terms of total space required, update per change, and several important queries.

2.7 Minimal Support Needed for TSQL2

The preceding discussed in general terms how a conventional DBMS could be extended to provide temporal support. In the remainder of this chapter, we describe the minimal changes needed by each component of the architecture to support a specific temporal query language: TSQL2.

Note that the precompiler and host language compiler are largely inde-pendent of the database query language-they require only small changes to support temporal literal/timestamp conversion. For each of the remain-ing components, the data dictionary and data files, as well as those within the DBMS proper, we describe the minimal modifications needed by these components to support TSQL2 queries.

Data Dictionary and Data Files

The data dictionary and data files contain the database, the actual data managed by the DBMS. The data dictionary records schema information such as file structure and format, the number and types of attributes in a table, integrity constraints, and associated indexes. The data files contain the physical tables and access paths of the database.

For a minimal extension, the data files require no revision. We can store tuple-timestamped temporal tables in conventional tables, where the time-stamp attributes are stored as explicit atomic attributes. However, the data dictionary must be extended in a number of ways to support TSQL2. The most significant extensions involve schema versioning, multiple granularities, and vacuuming.

For schema versioning, the data dictionary must record, for each table, all of its schemas and when they were current. The data files associated with a schema must also be preserved. This is easily accomplished by making a transaction-time table recording the schemas for a single table. The trans-action time associated with a tuple in this table indicates the time when the schema was current.

Multiple granularities are associated in a lattice structure specified at system generation time. A simple option is to store the lattice as a data structure in the data dictionary. Alternatively, if the lattice is fixed (i.e., new granularities will not be added after the DBMS is generated), then the lattice can exist as a separate data structure outside of the data dictionary.

Vacuuming specifies what information should be physically deleted from the database. Minimally, this requires a timestamp, the cutoff time, to be stored for each transaction-time or bitemporal table cataloged by the data dictionary. The cutoff time indicates that all data current in the table before the value of the timestamp has been physically deleted from the table.

DDL Compiler

The DDL compiler translates TSQL2 CREATE and ALTER statements into executable transactions. Each of these statements affects both the data dictionary and the data files. The CREATE statement adds new definitions, of either tables or indexes,

to the data dictionary and creates the data files containing the new table or index.The ALTER variants change an existing schema by updating the data dictionary, and possibly updating the data file containing the table. Numerous changes are needed by the DDL compiler, but each is straight-forward and extends existing functionality in small ways. First, the syntactic analyzer must be extended to accommodate the extended TSQL2 syntax for each of the CREATE and ALTER statements. The semantic analyzer must be extended in a similar manner, for example, to ensure that an existing table being transformed into a valid-time state table with the **ADD** VALID **STATE** command is not already a valid-time table.

Query Compiler

The query compiler translates TSQL2 data manipulation language (DML) statements into an executable, and semantically equivalent, internal form called the query execution plan. As with the DDL compiler, each phase of the query compiler-syntactic analysis, semantic analysis, and query plan generation-must be extended to accommodate TSQL2 queries.We use the model that the initial phase of the compilation, syntactic analysis, creates a tree-structured query representation, which is then refer-enced and augmented by subsequent phases. Abstractly, the query compiler performs the following steps:

" Parse the TSQL2 query. The syntactic analyzer, extended to parse the TSQL2 constructs, produces an internal representation of the query, the parse tree.

" Semantically analyze the constructed parse tree. The parse tree produced by the syntactic analyzer is checked for types and other semantic constraints, and simultaneously augmented with semantic information.

" Lastly, a query execution plan, essentially an algebraic expression that is semantically equivalent to the original query, is produced from the augmented parse tree by the query plan generator.

The minimal changes required by the query compiler are summarized as follows:

" The syntactic and semantic analyzers must be extended to support TSQL2.
" The query execution plan generator must be extended to support the extended TSQL2 algebra, including the new coalescing, join, and slic-ing op-

erations. In a minimally extended system, it may be acceptable to use existing algebraic equivalences for optimization, even with the extended operator set. Such an approach preserves the performance of

conventional snapshot queries. Later inclusion of optimization rules for the new operators would be beneficial to the performance of temporal queries.

"* Support for vacuuming must be included in the compiler. Query modification, which normally occurs after semantic analysis and prior to query optimization, must be extended to include vacuuming support.

The need to extend the syntactic and semantic analyzers is self-evident and straightforward. Extending the query plan generator to use the extended algebra is also straightforward, assuming that temporal aspects of the query are not considered during query optimization. In the worst case, the same performance would be encountered when executing a temporal query on a purely snapshot database. Lastly, in order to support vacuuming, the query compiler, within its semantic analysis phase, must support automated query modification based on vacuuming cutoff times stored in the data dictionary.

Run-Time Evaluator

The run-time evaluator interprets a query plan produced by the query com-piler. The run-time evaluator calls the transaction and data manager to retrieve data from the data dictionary and data files.

We assume that the run-time evaluator makes no changes to the query plan as received from the query compiler; that is, the query plan, as gen-erated by the query compiler, is optimized and represents the best possible evaluation plan for the query. As such, the changes required for the run-time evaluator are small. In particular, since evaluation plans for any new operators have already been selected by the query compiler, the run-time evaluator must merely invoke these operations in the same manner as non-temporal operations. Additionally, evaluation algorithms for new temporal operators (e.g., coalescing) are similar to well-known algorithms for snapshot operators. For example, coalescing can be implemented with slightly modi-fied duplicate elimination algorithms, which have been studied thoroughly.

UNIT-III
COMPLEX QUERIES AND REASONING

The importance of complex queries in advanced database systems can-not be overstated. At the introduction of the relational model, powerful logic-based queries were primarily motivated by their importance for end users. Subsequently, a long experience with SQL and large-scale commercial applications has shown that powerful query languages are essential in modern databases that use distributed environments, parallel machines, and client/server architectures.

Since support for complex queries means support for complex reasoning on large databases, this line of database work is also tackling problems previ-ously addressed in research domains, such as knowledge representation, non-monotonic reasoning, and expert systems. The next three chapters provide a unified introduction to the complex field of database and knowledge-based systems. In Chapter 8, we revisit relational query languages and extend them with more powerful constructs such as recursion, complex objects, and flexible set aggregates. In Chapter 9, we discuss the implementation of these extended queries in deductive databases and SQL systems. Finally, in Chap-ter 10, we explore recent advances in nonmonotonic reasoning that provide a unified model for temporal reasoning, active databases, and nondeterministic queries.

The Logic of Query Languages

First-order logic provides a conceptual foundation for relational query lan-guages. This foundation was established from the very first introduction of the relational data model by E. F. Codd, who introduced the parallel notions of relational calculus and relational algebra. Relational calculus provides a logic-based model for declarative query languages; relational al-gebra provides its operational equivalent: safe queries in predicate calculus can be transformed into equivalent relational expressions, and vice versa. The transformation of a calculus expression into an equivalent relational al-gebra expression represents the first step in efficient query implementation and optimization.

However, relational calculus has limited expressive power and cannot express many important queries, such as transitive closures and generalized aggregates. This situation has led to the design of more powerful logic-based languages that subsume relational calculus. First among these is the rule-based language Datalog, which is the focus of a large body of research and also of this chapter.

3.1 Datalog

In a Datalog representation, the database is viewed as a set of facts, one fact for each tuple in the corresponding table of the relational database, where the name of the relation becomes the predicate name of the fact. For instance, the facts in Example 3.2 correspond to the relational database of Example 3.1.

Example 3.1 **A** relational database about students and the courses they took

student

Name	*Major*	*Year*
Joe Doe	cs	senior
Jim Jones	cs	junior
Jim Black	ee	junior

took

Name	*Course*	*Grade*
Joe Doe	cs123	2.7
Jim Jones	csl01	3.0
Jim Jones	cs143	3.3
Jim Black	cs143	3.3
Jim Black	csl01	2.7

Example 3.2 The Datalog equivalent of Example **3.1**

```
student('Joe Doe', cs, senior).
student('Jim Jones', cs, junior).
student('Jim Black', ee, junior).
took('Joe Doe', cs123, 2.7).
took('Jim Jones', cs101, 3.0).
took('Jim Jones', cs143, 3.3).
took('Jim Black', cs143, 3.3).
took('Jim Black', cslO1, 2.7).
```

A *fact* is a logical predicate having only constants (i.e., no variables) as its arguments. We will use the accepted convention of denoting constants by tokens that begin with lowercase characters or numbers, while denoting variables by tokens that begin with uppercase. Thus, in a predicate such as

took(Name, cs143, Grade)

Name and Grade denote variables, while cs143 denotes a constant. How-ever, tokens in quotes, such as 'Jim Black', denote constants. Also, Name, cs143, and Grade are, respectively, the first, second, and third argument of the ternary predicate took. Both student and took are three-argument predicates, or equivalently, ternary predicates, or predicates with arity 3.

Rules constitute the main construct of Datalog programs. For instance, Example 3.3 defines all students at the junior level who have taken csl01 and cs143. Thus, firstreq(Name) is the head; student (Name, Major, junior), took(Name, csl01, Gradel), and took(Name, cs143, Grade2) are, re-spectively, the first, second, and third goal of the rule. Together, these three goals form the body of the rule.

Example 3.3 Find the name of junior-level students who have taken both csl01 and cs143

```
firstreq(Name) +- student (Name, Major, junior), took(Name, cst01, Gradel), took(Name, cs143, Grade2).
```

The commas separating the goals stand for logical conjuncts. Therefore, the order in which the goals appear in the rule is immaterial. Since the commas separating the goals stand for logical AND, the symbols **"A"** and **"&"** are often used in their place. Another common notational variation is the use of the symbol ":-" instead of the arrow to separate the head from the body of the rule.

A logical disjunct is represented via multiple rules with the same head predicate (i.e., sharing the same predicate name and arity). Thus, to find those juniors who took either course cs131 or course cs151, with grade better than 3.0, we would write the following:

Example 3.4 Junior-level students who took course cs131 or course cs151 with grade better than **3.0**

```
scndreq(Name)  -- took(Name, csl3L, Grade), Grade> 3.0, student(Name, Major, junior).
scndreq(Name)  - took(Name, csli, Grade), Grade> 3.0, student(Name, -, junior).
```

Observe that in the first rule of Example 3.4, the variable Maj or occurs only once; therefore, it can be replaced with the symbol "_", which is called an anonymous variable, and stands for a uniquely named variable that does not appear anywhere else in the rule (see the second rule of Example 3.4).

The set of rules having as their heads a predicate with the same name p is called the definition of p.

Thus, the definition of a derived predicate is similar to the definition of a virtual view in relational databases.

The meaning of such a definition is independent of the order in which these rules are listed, and independent of the order in which the goals appear in the rules.

Table 3.1 displays the corresponding nomenclatures of Datalog and the relational model.

Therefore, *base predicates* correspond to database relations and are defined by the database schema, while *derived predicates* are defined by rules.

It is also common to use the terms extensional *database* and intensional *database* to refer to base predicates and derived predicates, respectively.

In deductive databases, the assumption normally made is that these two form disjoint sets: that is, base predicates never appear in the heads of rules.

Datalog	Relational *Model*
Base predicate	Table or relation
Derived predicate	View
Fact	Row or tuple
Argument	Column or attribute

Table 3.1: The terminology of Datalog versus the relational model

Since rules are merely definitional devices, concrete Datalog programs also contain one or more query goals to specify which of the derived relations must actually be computed. Query goals can have different forms. A query that contains no variables is called a boolean query or a *closed* query; the answer to such a query is either yes or no. For instance,

?firstreq('Jim Black')

is a closed query with answer yes or no depending on whether 'Jim Black' has satisfied the first requirement. On the other hand, consider the goal

?f irstreq(X)

Since X is a variable, the answer to this query is a (possibly empty) set of facts for the students who satisfy the first requirement, as follows:

firstreq('Jim Jones')
firstreq('Jim Black')

In general, query goals will mix variables and constants in their arguments. Rules represent a powerful formalism from both theoretical and practical viewpoints. Their practical appeal follows from the ability to view goals in rules as search patterns. For instance, in the second rule of Example 3.4, we are searching for took tuples with cs151 as their second argument, and a grade greater than 3.0. Also, we are looking for the pattern junior in the third column of student, where the first attribute in this tuple is identical to the first value in the tuple of took, since all occurrences of the same variable in a rule must be assigned the same value.

The scope of variables, however, is local to rules, and identically named variables in different rules are considered independent.

The second important benefit of the Datalog formalism is its ability to break up the problem into smaller subproblems, each expressed by simple rules. Thus, complex patterns of computation and logical decisions can be achieved through rather simple Datalog rules that are stacked one upon another in a rich semantic structure.

For instance, say that in order to take the individual-study course cs298, a junior must have satisfied both requirements. Then we can simply write the following:

Example 3.4 Both requirements must be satisfied to enroll in cs298

```
req-cs298(Name)  +-   f irstreq(Name), scndreq(Name).
```

Therefore, derived relations can be used as goals in rules in the same fashion as database relations.

Datalog rules discussed so far are nonrecursive rules without negation. Additional expressive power can be achieved by allowing recursion and nega-tion in Datalog. We will next discuss negation; we discuss recursion later in this chapter.

Negation in Datalog rules can only be applied to the goals of the rule. Negation can never be used in the heads of rules. For instance, in Example 3.6, the second goal of the second rule is negated. This rule is meant to compute junior students who did not take course cs143.

Example 3.6 Junior-level students who did not take course cs143

```
hastaken(Name, Course) +-    took(Name, Course, Grade).
lacks-cs143(Name) +-         student(Name, -, junior), -ihastaken(Name, cs143).
```

Thus, hastaken defines the courses completed by a student, independent of the final grade. Then, the second rule selects those students for whom the pattern cs143 does not appear in the second column.

A frequent use of negation is in conjunction with universally quantified queries that are often expressed by words such as "each" and "every." For instance, say we would like to express the following query: "find the senior students who completed all requirements for a cs major."

The universally quantified condition "all requirements must be satisfied" can only be expressed in Datalog by transforming it into an equivalent con-dition where universal quantification is replaced by existential quantification and negation. This transformation normally requires two steps.

The first step is that of formulating the complementary query. For the example at hand, this could be "find students who did not take some of the courses required for a cs major." This can be expressed using the first rule in Example 3.7. Having derived those senior students who are missing some required courses, as the second step, we can now reexpress the original query as "find the senior students who are NOT missing any requirement for a cs major." This corresponds to the second rule in Example 3.7.

Example 3.7 Find the senior students who completed all the re-quirements for the cs major: ?allreqsat(X)

```
reqm-issing(Name)   *-   student(Name,    ,senior),
                         req(cs, Course),

                         -7hastaken(Name, Course).
all-req-sat(Name)        student(Name, , senior), -ireqinissing(Name).
```

Turning a universally quantified query into a doubly negated existen-tial query is never without difficulty,

but this is a skill that can be mastered with some practice. Indeed, such a transformation is common in natural lan-guages, particularly in euphemistic nuances. For instance, our last sentence,
" ... is never without difficulty," was obtained by rephrasing the original sentence " ... is always difficult."

3.2 Relational Calculi

Relational calculus comes in two main flavors: the domain relational calculus (DRC) and the tuple relational calculus (TRC). The main difference between the two is that in DRC variables denote values of single attributes, while in TRC variables denote whole tuples.

For instance, the DRC expression for a query ?firstreq(N) is

*{(N) 13Gl(took(N, cslO**1,** G_1)) A 3G $_2$ (took(N, cslS3, G_2)) A*
IM(student(N, M, junior)) **}**

The query ?scndreq(N) can be expressed as follows:

{(N) I *EG, IM(took(N, **cs131,** G)* **A** *G > 3.0* **A** *student(N,M,junior))* V ***3G,** 3M(took(N, **cs151, G)*** A ***G***
> 3.0 A student(N, M,junior))}

There are obvious syntactic differences that distinguish DRC from Da-talog, including the use of set definition by abstraction instead of rules.
Furthermore, DRC formulas contain many additional constructs such as ex-plicit quantifiers, nesting of parentheses, and the mixing of conjunctions and disjunctions in the same formula.

Negation and universal quantification are both allowed in DRC. There-fore, the query ?all-req-sat(N) can be expressed either using double nega-tion, or directly using the universal quantifier as shown in Example 3.3. This formula also features the implication sign **-+,** where p **--+** q is just a shorthand for **-,p** V **q.**

Example 3.8 Using a universal quantifier to find the seniors who completed all cs requirements

{(N)I]M(student(N, *M, senior))* **A**
VC(req(cs, C) -+]G(took(N, C, G))} **(3.1)**

The additional syntactic complexity of DRC does not produce a more powerful language. In fact, for each domain predicate calculus expression there is an equivalent, nonrecursive Datalog program. The converse is also true, since a nonrecursive Datalog program can be mapped into an equivalent DRC query.
Relational calculus languages are important because they provide a link to commercial database languages. For instance, Query-By-Example (QBE) is a visual query language based on DRC. However, languages such as QUEL and SQL are instead based on TRC.In TRC, variables range over the tuples of a relation. For instance, the TRC expression for a query ?firstreq(N) is the following:

Example 3.9 The TRC equivalent of the query ?firstreq(N) in Example **3.3**

{(t[1])I *lu3s(took(t)* A took(u) A student(s) A t[2] = *csl01* A u[2] = *cs13* A t[l] = u[1] A

s[3] =junior A *s[1] = t[1])}*

In Example 3.9, t and s are variables denoting, respectively, tuples in took and student. Thus, t[1] denotes the first component in t (i.e., that corresponding to attribute Name); t[2] denotes the Course value of this tuple. In general, if **ji,.--** ,-jn denote columns of a relation R, and t E R, then we will use the notation *t[j ,...,j,-]* to denote the n-tuple *(t[jl],...,t[jn]).*

The main difference between DRC and TRC is that TRC requires an explicit statement of equality, while in DRC equality is denoted implicitly by the presence of the same variable in different places. For instance, in

Example 3.9, the explicit conditions t[1] = u[1] and s[1] = t[1] are needed to express equality joins. Once again, however, these differences do not change the power of the language: TRC and DRC are equivalent, and there are mappings that transform a formula in one language into an equivalent one in the other.

3.3 Relational Algebra

Datalog rules and DRC or TRC formulas are declarative logic-based lan-guages, but relational algebra (RA) is an operator-based language. However, formulas in logical languages can be implemented by transforming them into equivalent RA expressions.

The main operators of relational algebra can be summarized as follows:

1 Union. The union of relations R and **S,** denoted *R* U ***S,*** is the set of tuples that are in R, or in S, or in both. Thus, it can be defined using TRC as follows:

RUS- {tit e RVt ***E*** S}

This operation is defined only if R and S have the same number of columns.

2. *Set* difference. The difference of relations R and **S,** denoted R - ***S,*** is the set of tuples that belong to R but not to S. Thus, it can be defined as follows: (t - r denotes that both t and r have n components and *t[1]* = r[1] A ... A t[n] - r[n]):

R - S = {tlt E *R* **A** -,3r(r **E** *S* ***A*** t - *r)}*

This operation is defined only if R and S have the same number of columns (arity).

3. Cartesian product. The Cartesian product of R and S is denoted R *x S. R x* ***S*** = *It* I(3r **E** *R) (Is* ***C S)*** *(t[1,.*

.. ,n] = r At[n **+ 1,.. .,** n **+m]** -- *s)}I*

If R has n columns and S has m columns, then R *x* S contains all the possible m + n tuples whose first m components form a tuple in R and the last n components form a tuple in S. Thus, R *x* S has m + n columns and JRI *x* **ISI** tuples, where R1 and **ISI** denote the respective cardinalities of the two relations.

" Projection. Let R be a relation with n columns, and L = $1,..., $n be a list of the columns of R. Let *L'* be a sublist of L obtained by

eliminating some of the elements, and (2) reordering the remaining ones in an arbitrary order. Then, the projection of R on columns *L,* denoted 7L,, is defined as follows:

$$7rn\ \mathrm{R} = \{r[L']\ \boldsymbol{I}\ \mathrm{r}\ \boldsymbol{C}\ R\}$$

R *Selection. UFR* denotes the selection on R according to the selection formula F, where F obeys one of the following patterns:

3.4 Syntax and Semantics of Datalog Languages

Following the general introduction to logic-based query languages in the previous sections, we can now present a more formal definition for such languages. Thus, we relate the syntax and semantics of query languages to those of first-order logic. Then, we introduce the model-theoretic semantics of Datalog pro

grams and present a fixpoint theorem that provides the formal link to the bottom-up implementation of such programs.

3.4.1 Syntax of First-Order Logic and Datalog

First-order logic follows the syntax of context-free languages. Its alphabet consists of the following:

a. *Constants.*
b. Variables: In addition to identifiers beginning with uppercase, x, y, and z also represent variables in this section.
c. Functions, such as *f(tl, ... ,* t.), where f is an n-ary functor and *ti,* **...- ,** tn are the arguments.
d. *Predicates.*
e. *Connectives:* These include basic logical connectives V, A, **-7,** and the implication symbols +-, -+, and ++.
f. Quantifiers: **]** denotes the existential quantifier and V denotes the universal quantifier.
g. Parentheses and punctuation symbols, used liberally as needed to avoid ambiguities.

Terms, atoms, and formulas that contain no variables are called ground.

Example 3.25 Well-formed formulas in first-order logic

$$]Gl(took(N, csl01, \boldsymbol{Gi}))\ \mathrm{A}\ IG_2\ (took(N,cs143, G_2))\ \mathbf{A}\ IM(student(N, M,\ \text{junior})) \quad (3.4)$$

$$\boldsymbol{3N,}\ ]M(student(N, M,\ \text{senior})\ \mathrm{A}\ VC(req(cs, C)\ \text{-+}\ 3G(took(N, C, G))))$$

$$\mathrm{VxVyVz}\ (p(x, z)\ \mathrm{V}\ \acute{y}q(x, y)\ \mathbf{V}\ \text{-ir}(y, z)) \quad (3.6)$$

$$\mathrm{VxVy}\ (\text{-}ip(x,y)\ \mathrm{V}\ q(f(x,y),a)) \quad (3.7)$$

A WFF F is said to be a *closed* formula if every variable occurrence in F is quantified. If F contains some variable x that is not quantified, then x is said to be a (quantification-) free variable in F, and F is not a closed formula. The variable N is not quantified in the first formula in Example 3.25 (3.4), so this formula is not closed. The remaining three WFFs in Example 3.25 are closed.

positive and negated atoms, whose every variable is universally quantified. A clause is called a definite clause if it contains exactly one positive atom and zero or more negated atoms. Thus a definite clause has the form

Vxl, ... , *Vxs* (A V-Bi V ... V -Bn)

Since F <- *G _ F* V -,*G,* the previous clause can be rewritten in the standard rule notation:

A - *B1,* **... ,** *Bn.*

A is called the head, and B_1, ... , *Bn,* is called the *body* of the rule.

In **Example** 3.25, only the WFFs 3.6 and 3.7 are clauses and are written as follows:

Example 3.26 The rule-based representation of clauses **3.6** and **3.7**

p(x,z) **+-** ***q(x,y),r(y,z)***

q(f(x, y), a) *- *p(x, **y).***

A definite clause with an empty body is called a unit clause. It is custom-ary to use the notation "A." instead of the more precise notation "A -- ." for such clauses. A *fact* is a unit clause without variables (see Example 3.27).

Example 3.27 **A** unit clause (everybody loves himself) and three facts

loves(X, X).
loves(marc, mary).
loves(mary, tom).
hates(marc, tom).

Definition 3.4 A *positive logic program is a set of* definite clauses.

We will use the terms definite clause program and *positive* program as synonyms.

3.3.2 Semantics

Positive logic programs have a very well defined formal semantics since al-ternative plausible semantics proposed for these programs have been shown to be equivalent. More general programs (e.g., those containing negation) are less well behaved and more complex in this respect and will be discussed in later chapters. For the rest of this chapter, the word "program" simply means a positive logic program (i.e., a set of definite clauses).

Implementation of Rules and Recursion

3.4 Rule-Rewriting Methods

The grandma predicate can be computed using the following relational alge-bra expression:

GRANDMA = 7r$,$ ((FATHER U *MOTHER)* M$1=$2 *MOTHER)*
3 2

which is the result of replacing selections on Cartesian products with equiv-alent joins in the RA expression produced by Algorithm 3.2. Then the answer to the query goal ?grandma(marc, GM) is *0r$ marc GRANDMA.* But

2

this approach is inefficient since it generates all pairs grandma/grand-child, even if most of them are later

discarded by the selection *9$2=marc.* A better
approach is to transform the original RA expression by pushing selection into the expression as is currently done by query optimizers in relational databases. Then we obtain the equivalent expression:

7r$ $_3$,S_2 ((_$2=maxcFATHER U *a$2=,macMOTHER) N91=92 MOTHER)*

In the RA expression so obtained, only the parents of marc are selected from the base relations mother and father and processed through the rest of the expression. Moreover, since this selection produces a binary relation where all the entries in the second column are equal to marc, the projection 7r$1 could also be pushed into the expression along with selection.
The optimization performed here on relational algebra can be performed directly by specializing the original rules via an SLD-like pushing of the query constants downward (i.e., into the rules defining the goal predicate); this produces the following program, where we use the notation X/a to denote that X has been instantiated to a:

Example 3.7 Find the grandma of marc

```
?grandma(GM,
marc)
grandma(Old, Young/marc)  +-   parent (Mid, Young/marc),
                               mother(Old, Mid).
parent(F, Cf/marc)        *-   father(F, Cf/marc).
parent (M,Cm/marc)        +-   mother(M, Cm/marc).
```

Thus, the second argument in the predicate parent is set equal to the constant marc.

3.4.1 Left-Linear and Right-Linear Recursion

If, in Example 3.6, we need to compute all the anc pairs, then a bottom-up approach provides a very effective computation for this recursive predicate. However, consider the situation where the goal contains some constant; for example, say that we have the query ?anc(tom, Desc). As in the case of nonrecursive rules, we want to avoid the wasteful approach of generating all the possible ancestor/person pairs, later to discard all those whose first component is not tom. For the recursive anc rule of Example 3.6, we can observe that the value of **Old** in the head is identical to that in the body; thus we can specialize our recursive predicate to anc(tom, **-)** throughout the fixpoint computation. As previously discussed, Prolog performs this operation during execution. Most deductive databases prefer a compilation-oriented approach where the program is compiled for a query form, such as anc($Name, X). The dollar sign before Name denotes that this is a deferred constant, i.e., a parameter whose value will be given at execution time. Therefore, deferred constants are treated as a constant by the compiler, and the program of Example 3.8 is rewritten using $Name as the first argument of anc.
Transitive-closure-like computations can be expressed in several equiv-alent formulations; the simplest of these use recursive rules that are either left-linear or right-linear. The left-linear version of anc is that of Example 3.6. Consider now the right-linear formulation of ancestor:

Example 3.9 Right-linear rules for the descendants of tom

```
anc(Old, Young) <--  parent(Old, Young).
anc(Old, Young) +-   parent(Old, Mid),anc(Mid,  Young).
```

With the right-linear rules of Example 3.9, the query ?anc($Name, X) can no longer be implemented by

specializing the rules. (To prove that, say that we replace **Old** with the constant $Name = tom; then, the transitive closure cannot be computed using parent(tom, Mid), which only yields children of tom, while the grandchildren of tom and their children are also needed.)

While it is not possible to specialize the program of Example 3.9 for a query ?anc($Name,X), it is possible to transform it into an equivalent pro-gram for which such a specialization will work. Take, for instance, the right-linear program of Example 3.9; this can be transformed into the equivalent left-linear program of Example 3.6, on which the specialization approach can then be applied successfully. While recognizing the equivalence of programs is generally undecidable, many simple left-linear rules can be detected and

[1]As a further improvement, the constant first argument might also be dropped from the recursive predicate. Transformed into their equivalent right-linear counterparts. Symmetric conclusions follow for the left-linear program of Example 3.6, which, for a query such as ? anc(Y, SD), is transformed into its right-linear equivalent of Exam-ple 3.3. Techniques for perfoming such transformations will be discussed in Section 3.6.

After specialization, left-linear and right-linear rules can be be supported efficiently using a single fixpoint computation. However, more complex re-cursive rules require more sophisticated methods to exploit bindings in query goals. As we shall see in the next section, these methods generate a pair of fixpoint computations.

3.4.2 Magic Sets Method

To illustrate the basic idea behind magic sets, let us first consider the fol-lowing example, consisting of two non recursive rules that return the names and addresses of senior students:

Example 3.10 Find the graduating seniors and the addresses of their parents

```
snr par-add(SN, PN, Paddr) <-   senior(SN),parent(PN, SN), address(PN, Paddr).
        senior(SN)  --          student(SN,_, senior),      graduating(SN).
```

A bottom-up computation on the rules of Example 3.10 determines grad-uating seniors, their parents, and the parents' addresses in an efficient man-ner. But, say that we need to find the address of a particular parent, for example, the address of Mr. Joe Doe, who has just called complaining that he did not get his invitation to his daughter's graduation. Then, we might have the following query: ?snr par-add (SN, 'Joe Doe', Addr). For this query, the first rule in Example 3.10 can be specialized by letting PN = 'Joe Doe'. Yet, using a strict bottom-up execution, the second rule still generates all names of graduating seniors and passes them up to the senior(SN) of the first rule. An optimization technique to overcome this problem uses an aux-iliary "magic" relation computed as follows:

Example 3.11 Find the children of Joe Doe, provided that they are graduating seniors

```
snr par-add-q('Joe    Doe').
m.senior(SN)  +-      snrparaddq(PN), parent (PN, SN).
```

The fact snr par-add-q('Joe Doe') stores the bound argument of the original query goal. This bound argument is used to compute a value of SN that is then passed to m.senior(SN) by the bottom-up rule in Example 3.11, emulat-ing what the first rule in Example 3.10 would do in a top-down computation. We can now improve the second rule of Example 3.10 as follows:

Example 3.12 Restricting search via magic sets

```
senior(SN)  *-  m.senior(SN),
                student(SN, -, senior), graduating(SN).
```

Therefore, the bottom-up rules of Example 3.12 are designed to emulate the top-down computation where the binding is passed from SN in the head to the first goal of parent. This results in the instantiation of SN, which is then passed to the argument of senior.

The "magic sets" notion is very important for those recursive predicates that are not amenable to the specialization treatment used for left-linear and right-linear rules. For instance, the recursive rule in Example 3.13 is a linear rule that is neither left-linear nor right-linear.

Example 3.13 People are of the same generation if their parents are of the same generation

```
?sg(marc, Who).
        sg(X, Y) +-   parent(XP, X), sg(XP, YP), parent(YP, Y).
        sg(A, A).
```

The recursive rule here states that X and Y are of the same generation if their respective parents XP and YP also are of the same generation. The exit rule sg(X, X) states that every element of the universe is of the same generation as itself. Obviously this rule is unsafe, and we cannot start a bottom-up computation from it. However, consider a top-down computa-tion on these rules, assuming for simplicity that the fact parent(tom, marc) is in the database. Then, the resolvent of the query goal with the first rule is +- parent(XP, marc), sg(XP, YP), parent(YP, Y). Then, by unifying the first goal in this list with the fact parent(tom, marc), the new goal list becomes +- sg(tom, YP),parent(YP, Y). Thus, the binding was passed from the first argument in the head to the first argument of the recursive predicate in the body. Now, the recursive call unfolds as in the previous case, yielding the parents of tom, who are the grandparents of marc. In summary, the top-down computation generates all the ancestors of marc using the recur-sive rule. This computation causes the instantiation of variables X and XP,while Y and YP remain unbound. The basic idea of magic sets is to emulate this top-down binding passing using rules to be executed in a bottom-up fashion. Therefore, we can begin by restricting our attention to the bound arguments and use the following rule: sg(X) +- parent (XP, X), sg(XP). Then, we observe that the top-down process where bindings are passed from X to XP through parent can be emulated by the bottom-up execution of the magic rule m.sg(XP) +- m.sg(X), parent(XP, X); the rule is constructed from the lastone by exchanging the head with the recursive goal (and adding the prefix "im."). Finally, as the exit rule for the magic predicate, we add the fact m.sg(marc), where marc is the query constant.

In summary, the magic predicate m.sg is computed as shown by the first two rules in Example 3.14. Example 3.14 also shows how the original rules are rewritten with the addition of the magic goal m.sg to restrict the bottom-up computation.

Example 3.14 The magic sets method applied to Example **3.13**

```
m.sg(marc).
m.sg(XP)  --    m.sg(X), parent (XP, X).
sg'(X, X) *-    m.sg(X).

sg'(X, Y) +-    parent (XP, X), sg'(XP, YP), parent (YP, Y), m.sg(X).
?sg'(marc, Z).
```

Observe that, in Example 3.14, the exit rule has become safe as a result of the magic sets rewriting, since only people who are ancestors of marc are considered by the transformed rules. Moreover, the magic goal in the recursive rule is useful in narrowing the search because it eliminates people who are not ancestors of marc.

Following our strict stratification approach, the fixpoint for the magic predicates will be computed before that of the modified rules. Thus, the magic sets method can be viewed as an emulation of the top-down computa-tion through a cascade of two fixpoints, where each fixpoint is then computed efficiently using the differential fixpoint computation.The fixpoint computation works well even when the graph representing parent is a directed acyclic graph (DAG) or contains directed cycles. In the case of a DAG, the same node and its successors are visited several times using SLD-resolution. This duplication is avoided by the fixpoint computation, since every new result is compared against those previously memorized. In the presence of directed cycles, SLD-resolution flounders in an infinite loop, while the magic sets method still works.An additional virtue for the magic sets method is its robustness, since the method works well in the presence of multiple recursive rules and even non-linear rules (provided that the binding passing property discussed in Section 3.6 holds).One problem with the magic sets method is that the computation per-formed during the first fixpoint might be repeated during the second fixpoint. For the example at hand, for instance, the ancestors of marc are computed during the computation of ms.sg and revisited again as descendants of those ancestors in the computation of sg'. The counting method and the supple-monetary magic sets technique discussed next address this problem.

3.4.3 The Counting Method

The task of finding people who are of the same generation as marc can be expressed as that of finding the ancestors of marc and their levels, where marc is a zero-level ancestor of himself, his parents are first-generation (i.e., first-level) ancestors, his grandparents are second-generation ancestors, and so on. This computation is performed by the predicate sg up

The counting method mimics the original top-down SLD-resolution to such an extent that it also shares some of its limitations. In particular, cycles in the database will throw the rewritten rules into a perpetual loop; in fact, if sgup(J, XP) is true and XP is a node in the loop, then sg up(J + K, XP), with K the length of the cycle, holds as well.

Another problem with counting is its limited robustness, since for more complex programs, the technique becomes inapplicable or requires several modifications. For instance, let us revise Example 3.13 by adding the goal XP *$7*YP to the recursive rule, to avoid the repeated derivation of people who are of the same generation as themselves. Then, the rules defining sg up must be modified to memorize the values of XP, since these are needed in the second fixpoint. By contrast, the supplementary magic technique discussed next disregards the level information and instead relies on the systematic memorization of results from the first fixpoint, to avoid repeating the same computation during the second fixpoint.

3.4.4 Supplementary Magic Sets

In addition to the magic predicates, supplementary predicates are used to store the pairs bound-arguments-in-head/bound-arguments-in-recursive-goal produced during the first fixpoint. For instance, in Example 3.16, we compute spm.sg, which is then used during the second fixpoint computation, since the join of spm.sg with sg' in the recursive rule returns the memorized value of X for each new XP. Described, and in fact the two terms are often used as synonyms. Frequently, the magic predicate and the supplementary magic predicate are written in a mutually recursive form. Thus, for Example 3.16, we have the following rules:

Example 3.17 The magic and supplementary magic rules for **3.13**

```
m.sg(marc).
spm.sg(X, XP)  +- m.sg(X), parent (XP, X)
m.sg(XP) +-        spm.sg(X, XP).
```

To better understand how the method works, let us revise the previous example. Say that we only want to search up to *kth* generations where the parents and their children lived in the same state. Then, we obtain the following

program:

As illustrated by this example, not all the bound arguments are memo-rized. Only those that are needed for the second fixpoint are stored in the supplementary magic relations. In our case, for instance, St is not included.

Because of its generality and robustness, the supplementary magic tech-nique is often the method of choice in deductive databases. In fact, the method works well even when there are cycles in the underlying database. Moreover, the method entails more flexibility with arithmetic predicates. For instance, the expression KP = K - 1 is evaluated during the first fixpoint, where K is given and the pair (K, KP) is then memorized in the supplementary relations for use in the second fixpoint. However, with the basic magic sets method from the second fixpoint, K can only be computed from the values of KP taken from 6stsg(XP, KP, YP), provided that the equation KP = K - 1 is first solved for K. Since this is a simple equation, solving it is a simple task for a compiler; however, solving more general equations might either be very difficult or impossible. An alternative approach consists in using the arithmetic equality as is, by taking each value of K from the magic set and computing K - 1. However, this computation would then be repeated with no change at each step of the second fixpoint computation. The use of sup-plementary magic predicates solves this problem in a uniform and general way since the pairs K, KP are stored during the first fixpoint and then used during the second fixpoint.

The supplementary magic method can be further generalized to deal with nonlinear rules, including nonlinear rules as discussed in the next section (see also Exercise 3.7).

3.6 Compilation and Optimization

Most deductive database systems combine bottom-up techniques with top-down execution. Take for instance the fiat parts program shown in Example 3.3, and say that we want to print a list of part numbers followed by their weights using the following query: ?part-weight(Part, Weight). An execu-tion plan for this query is displayed by the rule-goal graph.

The graph depicts a top-down, left-to-right execution, where all the pos-sible unifications with rule heads are explored for each goal. The graph shows the names of the predicates with their bound/free adornments positioned as superscripts. Adornments are vectors of *f* or b characters. Thus, a *kth* char-acter in the vector being equal to b or f denotes that the *kth* argument in the predicate is respectively bound or free. An argument in a predicate is said to be bound when all its variables are instantiated; otherwise the argument is said to be free, and denoted by **f.**

3.6.1 Non-recursive Programs

The rule-goal graph for a program P is denoted rgg(P). The rule-goal graph for a non-recursive program is constructed as follows:

Algorithm **3.11** Construction *of the* rule-goal *graph rgg(P)* for a nonre-cursive program P.

" Initial *step:* The query goal is adorned according to the constants and deferred constants (i.e., the variables preceded by $), and becomes the root of *rgg(P).*

" *Bindings passing from goals to rule heads:* If the calling goal g unifies with the head of the rule r, with mgu **-y,** then we draw an edge (labeled with the name of the rule, i.e., r) from the adorned calling goal to the adorned head, where the adornments for h(r) are computed as follows:

(i) all arguments bound in g are marked bound in h(r)Y;
(ii) all variables in such arguments are also marked bound; and
(iii) the arguments in h(r)y that contain only constants or variables marked bound in (ii) are adorned b, while the others are adorned f.

For instance, say that our goal is g = p(f(Xi), 1-, Z 1 , a), and the head of r is h(r) = p(X 2 ,g(X 2 ,Y 2),Y 2 ,W 2). (If g and h(r) had variables in common, then a renaming step would be required here.) A most general unifier exists for the two: y = {X 2 /f(Xl), Y 1 /g(f(X 1),Y 2), Z 1/Y 2 , W2 /a}; thus, bindings might be passed from this goal to this head in a top-down execution, and the resulting adornments of the head must be computed. The unified head is h(r)•/= p(f(X1), g(f(Xi), Y2), Y2 , a). For instance, say that the goal was adorned pbffb; then variables in the first argument of the head (i.e., Xi) are bound. The resulting adorned head is pbffb, and there is an edge from pbffb to pbffb +-. But if the adorned goal is pfbfb, then all the variables in the second argument of the head (i.e.,X 1 , Y2) are bound. Then the remaining arguments of the head are bound as well. In this case, there is an edge from the adorned goal pfbfb to the adorned head pbbbb.

Left-to-right passing of bindings to goals: A variable X is bound after the ***nfh*** goal in a rule, if X is among the bound head variables (as for the last step), or if X appears in one of the goals of the rule preceding the ***nth*** goal.

The (n + ***1)th*** goal of the rule is adorned on the basis of the variables that are bound after the *nfth* goal. For simplicity of discussion, we assume that the rule-goal graph for a non-recursive program is a tree, such as that of Figure 3.1. Therefore, rather than drawing multiple edges from different goals to the same adorned rule head, we will duplicate the rule head to ensure that a tree is produced, rather than a DAG.

The rule-goal graph determines the safety of the execution in a top-down mode and yields an overall execution plan, under the simplifying assumption that the execution of a goal binds all the variables in the goal. The safety of the given program (including the bound query goal) follows from the fact that certain adorned predicates are known to be safe a priori. For instance, base predicates are safe for every adornment. Thus, part ^{f} is safe. Equality and comparison predicates are treated as binary predicates. The pattern $_{0}$ **bb** is safe for 0 denoting any comparison operator, such as < or >Moreover, there is the special case of **=bf** or **=fb** where the free argument consists of only one variable; in either case the arithmetic expression in the bound argument can be computed and the resulting value can be assigned $_{to}$ the free variable.

Definition 3.12 *Let P be* a program with rule-goal *graph rgg(P), where rgg(P) is* a tree. Then *P is safe if the following two conditions hold:*

a. Every leaf *node of rgg(P) is safe a* priori, and
b. every variable in every rule in *rgg(P) is* bound *after the last goal.*

Given a safe *rgg(P),* there is a simple execution plan to compute rules and predicates in the program. Basically, every goal with bound adornments generates two computation phases. In the first phase, the bound values of a goal's arguments are passed to its defining rules (its children in the rule-goal graph). In the second phase, the goal receives the values of the f-adorned arguments from its children. Only the second computation takes place for goals without bound arguments. Observe that the computation of the heads of the rules follows the computation of all the goals in the body. Thus, we have a strict stratification where predicates are computed according to the post order traversal of the rule-goal graph.

Both phases of the computation can be performed by a relational algebra expression. For instance, the set of

all instances of the bound arguments can be collected in a relation and passed down to base relations, possibly using the magic sets technique-resulting in the computation of semijoins against the base relations. In many implementations, however, each instance of bound arguments is passed down, one at a time, to the children, and then the computed values for the free arguments are streamed back to the goal.

3.6.2 Recursive Predicates

The treatment of recursive predicates is somewhat more complex because a choice of recursive methods must be performed along with the binding passing analysis.

The simplest case occurs when the goal calling a recursive predicate has no bound argument. In this case, the recursive predicate, say p, and all the predicates that are mutually recursive with it, will be computed in a single differential fixpoint. Then, we fall back into the treatment of rules for the nonrecursive case, where

step 3 of Algorithm 3.11 is performed assuming that rule heads have no bound argument

d. safety analysis is performed by treating the recursive goals (i.e., p and predicates mutually recursive with it) as safe a priori-in fact, they are bound to the values computed in the previous step.

When the calling goal has some bound arguments, then, a binding *passing analysis* is performed to decide which method should be used for the case at hand. After this analysis, the program is rewritten according to the method selected.

Figure 3.2 illustrates how the binding passing analysis is performed on recursive rules. The binding passing from a goal to the recursive rule heads remains the same as that used for the nonrecursive case (step 2 in Algorithm 3.11). There are, however, two important differences. The first is that we allow cycles in the graph, to close the loop from a calling recursive goal to a matching adorned head already in the graph. The second difference is that the left-to-right binding passing analysis for recursive rules is more restrictive than that used at step 3 of Algorithm 3.11; only particular goals (called chain goals) can be used.

An adorned goal **qYý** in a recursive rule r is called a chain goal when it satisfies the following conditions:

a. *SIP independence of recursive goals: q* is not a recursive goal (i.e., not the same predicate as that in the head of r, nor a predicate mutually recursive with q; however, recursive predicates of lower strata can be used as chain goals).

b. *Selectivity:* ***q'Y*** has some argument bound (according to the bound variables in the head of r and the chain goals to the left of q7).

The basic idea behind the notion of chain goals is that the binding in the head will have to reduce the search space. Any goal that is called with all its adornment free will not be beneficial in that respect. Also, there is no sideway information passing (SIP) between two recursive goals; bindings come only from the head through non recursive goals.

The algorithm for adorning the recursive predicates and rules constructs a set of adorned goals A starting from the initial query goal (or a calling goal) q that has adornment **-/,** where ***-y*** contains some bound argument.

Algorithm 3.13 Binding *passing analysis for recursive predicates*

Initially **A** - *{qY}, with* ***qY*** *the* initial *goal, where q is* a recursive predicate and ***-y*** *is* not a *totally free* adornment.

For each h **E A,** *pass the binding to the heads of* rules defining *q.*

For each recursive rule, determine the adornments *of its recursive goals (i.e., of q or* predicates mutually recursive *with q). If the last step* generated adornments not currently in **A,** *add them to* **A** and resume from *step 2. Otherwise halt.*

The calling goal g is said to have the binding *passing* property when A does not contain any recursive predicate with totally free adornment. In this case, we say that g has the unique binding passing property when A contains only one adornment for each recursive predicate.

When the binding passing property does not hold, then a totally free adornment occurs, and mutually recursive predicates must be computed as if the calling goal had no bound arguments. Otherwise, the methods described in the previous sections are applicable, and the recursive program is rewritten according to the method selected.

3.6.3 Selecting a Method for Recursion

For simplicity of discussion, we assume that the unique binding passing property holds and concentrate on the rewriting for the magic sets method, which can then be used as the basis for other methods.

Let ***qY*** **E A,** and r be a recursive rule defining q. Then, if the recursive rank of r is k, then there are k magic rules corresponding to r: one for each recursive goal in r. If p is one of these goals, then the head of the magic rule is named m.p, and has as arguments the arguments of p bound according to ***qy.*** The body of the magic rule consists of the following goals: the recursive goal m.q with the bound arguments in ***qY,*** and the chain goals of r.

The (one and only) exit rule for all the magic predicates is actually the fact m.g', where g' is obtained from the calling goal by eliminating its free arguments.

Finally, each original rule r is augmented with the addition of a magic goal as follows. Say that q is the head of r, ***qy*** **G** A, and q' is obtained from h(r) by eliminating all the arguments that are free (i.e., denoted by an f in **-y);** then, m.q' is the magic goal added to r.

The rewriting methods for supplementary magic predicates, and for the counting method, can be derived as simple modifications of the templates for magic sets. While the counting method is limited to the situation where we have only one recursive rule and this rule is linear, the other two methods are applicable whenever the binding passing property holds (see Exercise 3.7).

The magic sets method can also be used as the basis for detecting and handling the special cases of left-linear and right-linear rules. For instance, if we write the magic rules for Example 3.8, we obtain:

```
m.anc(tom).
m.anc(Old)  +-  m.anc(Old).
```

Obviously the recursive magic rule above is trivial and can be eliminated. Since the magic relation anc now contains only the value tom, rather than appending the magic predicate goal to the original rules, we can substitute this value directly into the rules. It is simple for a compiler to recognize the situation where the body and the head of the rule are identical, and then to eliminate the magic rule and perform the substitution. Consider now the application of the magic sets method to Example 3.3. We obtain

```
m.anc(tom).
  m.anc(Mid)  *-  parent(Old,  Mid),  m.anc(Old).  anc'(Old,  Young)  -
  m.anc(Old),parent(Old, Young). anc'(Old, Young) +- parent (Old, Mid), anc'(Mid,
  Young),
  rn.anc'(Old).
```

?anc'(tom, Young).

Observe that the recursive rule defining anc' here plays no useful role. In fact, the second argument of anc' (i.e., Young) is simply copied from the goal to the head of the recursive rule. Moreover, once this second argument is dropped, then this rule simply revisits the magic set computation leading back to torn. Thus, every value of Young produced by the exit rule satisfies the query. Once the redundant recursive rule is eliminated, we obtain the following program:

```
m.anc(tom).
m.anc(Mid) +-        m.anc(Old), parent (Old, Mid).
anc'(Old, Young)  *- m.anc(Old),parent (Old, Young).
?anc'(tom, Young).
```

In general, for the recursive rule to be dropped, the following two con-ditions must hold: **(1)** all the recursive goals in the recursive rule have been used as chain goals (during the binding passing analysis), and (2) the free arguments in the recursive goal are identical to those of the head. These are simple syntactic tests for a compiler to perform. Therefore, the transforma-tion between right-linear recursion and left-linear recursion can be compiled as a special subcase of the magic sets method.

3.6.4 Optimization Strategies and Execution Plan

Several variations are possible in the overall compilation and optimization strategy described in the previous sections. For instance, the requirement of having the unique binding passing property can be relaxed easily (see Exercise 3.9). The supplementary magic method can also be generalized to allow the passing of bindings between recursive goals in the same rule; however, the transformed programs so produced can be complex and inefficient to execute.

A topic that requires further research is query optimization. Most rela-tional databases follow the approach of estimating the query cost under all possible join orders and then selecting the plan with the least-cost estimate. This approach is not commonly used in deductive database prototypes be-cause of its prohibitive cost for large programs and the difficulty of obtaining reliable estimates for recursive predicates. Therefore, many systems use in-stead simple heuristics to select an order of execution for the goals. For instance, to select the next goal, precedence is given to goals that have more bound arguments and fewer unbound arguments than the other goals.

In other systems, the order of goal execution is that in which they appear in the rule (i.e., the Prolog convention also followed in the rule-goal graph of Figure 3.1). This approach leaves the control of execution in the hands of the programmer, with all the advantages and disadvantages that follow. A promising middle ground consists of using the optimization techniques of relational systems for simple rules and queries on base predicates, while letting the programmer control the execution of more complex programs, or predicates more remote from the base predicates.

Although different systems often use a different mix of recursive meth-ods, they normally follow the same general approach to method selection. Basically, the different techniques, each with its specific applicability preconditions, are ranked in order of desirability; the first applicable method in the list is then selected. Therefore, the binding passing property is tested first, and if this is satisfied, methods such as those for left-linear and right-linear recursion are tried first; then if these fail, methods such as magic sets and supplementary magic are tried next. Several other techniques have been proposed for recursion, and novel approaches and refinements are being pro-posed frequently-although it is often difficult to evaluate the comparative effectiveness of the different techniques.

An additional generalization that should be mentioned allows some arguments of a goal to remain uninstantiated after its execution. In this approach, variables not bound by the execution of the goal will need to be bound by later goals, or will be returned to the head of the rule, and then to the calling goal, as unbound variables.

In addition to the global techniques discussed above, various optimizations of a local and specialized nature can be performed on Datalog-like languages. One such technique consists in avoiding the generation of multi-ple bindings for existential variables, such as variables that occur only once in a rule. Techniques for performing intelligent backtracking have also been used; these can, for example, simulate multiday joins in a tuple at a time execution model. Therefore, many of the local optimization techniques used are specific to the low-level execution model adopted by the system; this, in turn, depends on many factors, including whether the system is primarily de-signed for data residing on secondary storage or data already loaded in main memory. These alternatives have produced the assortment of techniques and design choices explored by current deductive database prototypes.

3.7 Recursive Queries in SQL

The new SQL3 standards include support for recursive queries. For instance, the BoM program of Example 3.15 is expressible in SQL3, using the view construct as follows:

Example 3.20 Recursive views in SQL3

```
CREATE RECURSIVE view all-subparts (Major, Minor) AS SELECT PART
SUBPART
FROM assembly
UNION

SELECT all.Major assb.SUBPART
FROM all-subparts all, assembly assb
        WHERE all.Minor- assb.PART
```

The **SELECT** statement before **UNION** is obviously equivalent to the exit rule in Example 3.15, while the **SELECT** statement after **UNION** corre-sponds to the recursive rules. Therefore we will refer to them as *exit select* and recursive select, respectively.

Since all-subparts is a virtual view, an actual query on this view is needed to materialize the recursive relation or portions thereof. For instance, the query of Example 3.21 requests the materialization of the whole relation.

Example 3.21 Materialization of the view of Example **3.20**

```
SELECT *
FROM all-subparts
```

The WITH construct provides another way, and a more direct one, to express recursion in SQL3. For instance, a query to find all the super parts using 'top-tube' can be expressed as follows:

Example 3.22 Find the parts using ‘top-tube'

```
WITH RECURSIVE all-super(Major, Minor) AS ( SELECT PART,
```

```
        SUBPART
        FROM assembly
        UNION
        SELECT assb.PART, all.Minor
        FROM assembly assb, all-super all
        WHERE assb.SUBPART = all.Major
        )
SELECT *
WHERE Minor = 'top-tube'
```

3.2.6 Implementation of Recursive SQL Queries

The compilation techniques developed for Datalog apply directly to recursive SQL queries. For instance, the query of Example 3.21 on the view defined in Example 3.20 requires the materialization of the whole transitive closure, and can thus be implemented efficiently using the differential fixpoint Algorithm 3.4. Then, *TE(S')* and TR(S') are, respectively, computed from the exit select and the recursive select in Example 3.20. Here too, the computation of *TR(S')* - ***S'*** will be improved using the differential fixpoint technique. In fact, this step is simple to perform since there is only one recursive relation in the FROM clause of Example 3.20; therefore, this is a case of linear recursion. Thus, the recursive relation all-subparts in the FROM clause is replaced with 5all-subparts, which contains new tuples generated in the previous iteration of Algorithm 3.4. Consider now Example 3.22. This requires the passing of the condition Minor= 'top-tube' into the recursive SQL query defined using WITH. Now, the recursive select in Example 3.22 uses right-linear recursion, whereby the second argument of the recursive relation is copied unchanged by *TR.* Thus, the condition Minor = 'top-tube' can simply be attached unchanged to the WHERE clause of the exit select and the recursive select, yielding the following equivalent SQL program:

Example **3.23** Specialization of the query of Example **3.22**

```
WITH RECURSIVE all-super(Major, Minor) AS (SELECT PART,
      SUBPART
        FROM assembly
        WHERE SUBPART = 'top-tube'
        UNION
        SELECT assb.PART, all.Minor
        FROM assembly assb, all-super all
        WHERE assb.SUBPART = all.Major
      )AND all.Minor - 'top-tube'
SELECT *
```

However, say that the same query is expressed against the virtual view of Example 3.20, as follows:

```
SELECT *
FROM all-subparts
WHERE Minor = 'top-tube'
```

Since all-subparts is defined in Example 3.20 using left-linear recursion, the addition of the condition

Minor = 'top-tube' to the recursive select would not produce an equivalent query. Instead, the SQL compiler must trans-form the original recursive select into its right-linear equivalent before the condition Minor = 'top-tube' can be attached to the WHERE clause. The compilation techniques usable for such transformations are basically those previously described for Datalog.

UNIT – IV

Spatial, Text, and Multimedia Databases

.

The problem we focus on in this part is the design of fast searching methods that will search a database of spatial, text, or multimedia objects to locate those that match a query object, exactly or approximately. Objects can be 2-dimensional color images, gray-scale medical images in 2-D or **3-D** (e.g., MRI brain scans), 1-dimensional time series, digitized voice or music, video clips, and so on. A typical query by content would be "in a collection of color photographs, find ones with the same color distribution as a sunset photograph."

4.1 Secondary Keys

Access methods for secondary key retrieval have attracted much interest. The problem is stated as follows: Given a file, say, EMPLOYEE (name, salary, age), organize the appropriate indices so that we can answer efficiently queries on any and all of the available attributes. Classified the possible queries into the following classes, in increasing order of complexity:

1. *Exact* match query, when the query specifies all the attribute values of the desired record. For example,
   name = 'Smith' and salary = 40,000 and age -= 45
2. Partial match query, when only some of the attribute values are specified. For example
   salary = 40,000 and age = 45
3. Range queries, when ranges for some or all of the attributes are specified. For example
   35,000 < salary _< 45,000 and age = 45
4. *Boolean* queries:
   ((not name = 'Smith') and salary >_ 40,000) or age > 50

4.2 Inverted Files

This is the most popular approach in database systems. An inverted file on a given attribute (say, salary) is built as follows: For each distinct attribute value, we store

1. a list of pointers to records that have this attribute value *(postings list)*
2. optionally, the length of this list

The set of distinct attribute values is typically organized as a B-tree or as a hash table. The postings lists may be stored at the leaves, or in a separate area on the disk. Below Figure shows an index on the salary of an EMPLOYEE table. A list of unique salary values is maintained, along with the postings lists. Given indices on the query attributes, complex Boolean queries can be resolved by manipulating the lists of record pointers before accessing the actual records. Notice that indices can be created automatically by a relational DBMS, with the SQL command CREATES **INDEX.**

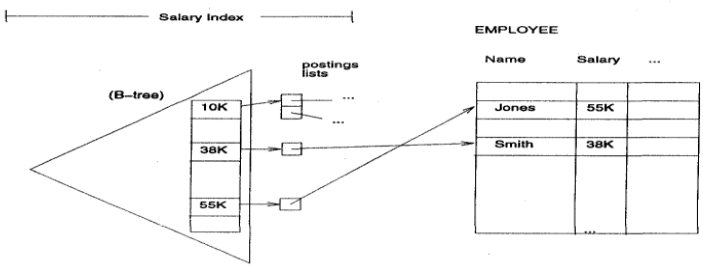


Illustration of inversion: a B-tree index on salary

4.3 Spatial access method

The secondary-key access methods, which handle queries on keys that may have duplicates (e.g., salary, or age, in an EMPLOYEE file). As mentioned, records with k numerical attributes can be envisioned as k-dimensional points. Here we examine spatial access methods, which are designed to handle multidimensional points, lines, rectangles, and other geometric bodies.

There are numerous applications that require efficient retrieval of spatial objects:

- Traditional relational databases, where, as we mentioned, records with k-attributes become points in k-D spaces.
- Geographic information systems (GISs), which contain, for example, point data, such as cities on a 2-dimensional map.
- Medical image databases with, for example, 3-dimensional MRI brain scans, which require the storage and retrieval of point sets, such as digitized surfaces of brain structures [21].
- Multimedia databases, where multidimensional objects can be represented as points in feature space [227, 157]. For example, 2-D color images correspond to points in (R,G,B) space (where R, G, and B are the average amount of red, green, and blue [152]).
- Time-sequence analysis and forecasting [452, 87], where k successive values are treated as a point in k-D space; correlations and regularities in this k-D space help in characterizing the dynamical process that generates the time series.
- Rule indexing in expert database systems [420] where rules can be represented as ranges in address space (e.g., "all the employees with salary in the range 10K-20K and age in the range 30-50 are entitled to specific health benefits").

4.4 1-D Time Series

Here the goal is to search a collection of (equal-length) time series, to find the ones that are similar to a desirable series. For example, in a collection of yearly stock price movements, we want to find the ones that are similar to IBM. For the rest of the section, we shall use the following notational conventions: If S and Q are two sequences, then

1. Len(S) denotes the length of S
2. *S[i :* j] denotes the subsequence that includes entries in positions I Through j
3. S[i] denotes the *ith* entry of sequence S
4. *D(S, Q)* denotes the distance of the two (equal-length) sequences S and Q

4.5 2-D Color Images

The goal is to study ways to query large on-line image databases using the images' content as the basis of the queries. Examples of the content we use include color, texture, shape, position, and dominant edges of image items and regions. Potential applications include medical ("give me other images that contain a tumor with a texture like this one"), photojournalism ("give me images that have blue at the top and red at the bottom"), and many others in art, fashion, cataloging, retailing, and industry.

4.6 Sub pattern Matching:

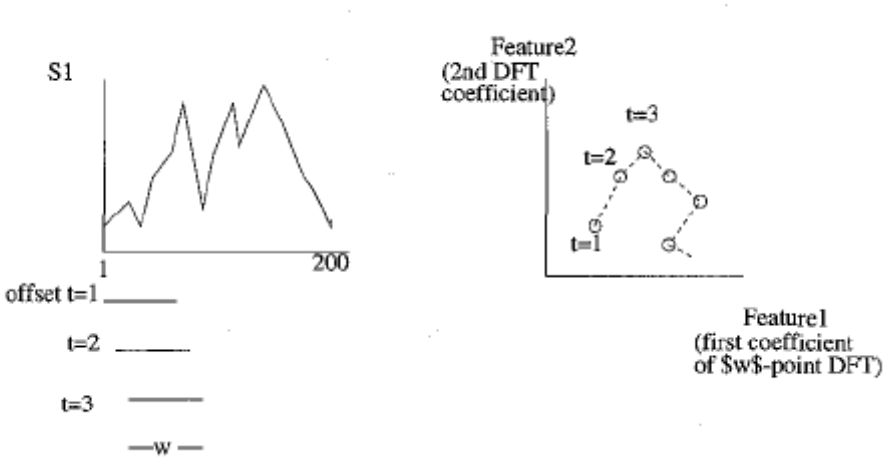
Up to now, we have examined the whole match case. The question is, Could we extend the philosophy of the quick-and-dirty test, so that we can handle subpattern matching queries? Let's focus on 1-D time series to illustrate the problem and the solution more clearly. Then, the problem is defined as follows:

1. We are given a collection of N sequences of real numbers ***S1, S2 ,*** *SN,* each one of potentially different length.
2. The user specifies query subsequence **Q** of length Len(Q) (which may vary) and the tolerance c, that is, the maximum acceptable dissimilarity (= distance).
3. We want to find quickly all the sequences Si (1 < i < N), along with the correct offsets k, such that the subsequence Si[k *: k* + Len(Q) - 1] matches the query sequence: *D(Q, Si[k :* k + Len(Q) - **1])** • E.

4.6.1 Sketch of the Approach-ST-Index:

Without loss of generality, we assume that the minimum query length is w, where w (Ž 1) depends on the application. For example, in stock price databases, analysts are interested in weekly or monthly patterns because shorter patterns are susceptible to noise [139]. Notice that we never lose the ability to answer shorter than w queries because we can always resort to sequential scanning.

Generalizing the reasoning of the method for whole matching, we use a sliding window of size w and place it at every possible position (offset), on every data sequence. For each such placement of the window, we extract the features of the subsequence inside the window. Thus, a data sequence of length Len(S) is mapped to a trail in feature space, consisting of *Len(S) -w* + 1 points: one point for each possible offset of the sliding window. BELOW Figure gives an example of a trail: Consider the sequence **S1,** and assume that we keep the first *k* = 2 features (e.g., the amplitude of the first and second coefficient of the w-point DFT). When the window of length w is placed at offset = 0 on ***S1,*** we obtain the first point of the trail; as the window slides over **S1,** we obtain the rest of the points of the trail.



UNIT - V
Uncertainty in Databases and Knowledge Bases

5.1 Uncertainty in Image Database:

For example, consider the problem of representing image content UNIT-II in a relational database. Consider a very simple relation called face that specifies which persons' faces are contained in which image files. Such a relation may have the schema

(File, Person, LLx, LLy, Ugx, URy)

and a simple instance of the face relation is shown below:

File	Person	LLx	LLy	URx	URy
im1.gif	John Smith	10	10	20	20
im1.gif	Mark Bloom	10	10	20	20
im1.gif	Mark Bloom	30	10	40	20
im1.gif	Ted Lewis	30	10	40	20
im2.gif	Mark Bloom	50	10	60	20
im2.gif	Ted Lewis	10	10	20	20
im3.gif	Lynn Bloom	10	10	20	20
im3.gif	Elsa Bloom	10	10	20	20

The attribute names may be interpreted as follows:

- File is the name of an image file (e.g., iml .gif).
- (LLx, LLy) and (URx, URy) specify the lower-left corner and the upperright corner of a rectangle (with sides parallel to the x- and y-axes) that bounds a particular person's face. Thus, in the above example, the first tuple indicates that there is a face (in iml. gif) in the rectangular region whose lower-left corner is at (10,10) and whose upper-right corner is at (20,20). Thus, the (LLx,LLy) and (URx,URy) components of any tuple uniquely capture a rectangle within the specified image.
- Person specifies the name of the person whose face occurs in the rectangle specified by a tuple in this relation. Thus, for instance, the first tuple in the face relation states that the person in the rectangular region whose lower-left corner is at (10,10) and whose upper-right corner is at (20,20) is John Smith.

5.2 Uncertainty in Temporal Database:

Often, a tuple in a relational database is time stamped with an interval of time. This often denotes the fact that the tuple was true at some time instant in that interval. For example, we may have a temporal relation called shipping that is maintained by a factory. This relation may have the schema

(Item, Destination).

When extended to handle temporal information, we may have a new additional attribute called ShipDate that denotes the date on which the item was shipped. The expanded shipping relation may contain the following tuples:

Item	*Destination*	*When*
widget-1	Boston	Jan. 1 ~ Jan. 7, 1996
widget-1	Chicago	Jan. 2, 1996
widget-2	Omaha	Feb. 1 ~ Feb. 7, 1996
widget-2	Miami	Feb. 18 ~ Feb. 21, 1996

The first tuple above says that the factory shipped an order of widget-1 to Boston sometime between January 1 and January 7 (inclusive). However, the precise date is unknown. Consider now the query "find all places to which widget-1 was shipped on or before January 5, 1996." As we will see below, some different answers are possible:

- If we were to assume that the probability of being shipped on any given day is equally likely, then there is a 5 probability that the Boston shipment was on or before January 5. Thus, we should return the answer

Destination	*Prob*
Boston	$\frac{5}{7}$
Chicago	1

- On the other hand, if we know that nothing is shipped from the factory over the weekend, then we know, since January 1, 1996, was a Monday, that the probability of the shipment going out between January 1 and January 5 is 1, and we should return the answer

Destination	*Prob*
Boston	1
Chicago	1

5.3 Uncertainty in DBs: A Null-Value Example

As you are probably aware, it is not always possible to associate a value with each and every column of each and every tuple in a given relation. For example, because of some unforeseen conditions (e.g., a coffee spill), the destination of a particular shipment may not be deductible from a given shipping invoice. However, the name of the intended recipient may be visible, leading the database administrator to conclude that the shipment was intended for one of the two factories of that company, located in New York and Denver.

The database administrator, after speaking to the shipping department, may conclude that most likely the shipment was intended for Denver (with 90% certainty). In this case, the following data may be entered into the database.

5.4 Fuzzy logic

In classical logic, there is a close correspondence between sets and logic. If F is a formula in such a logical language, then F denotes the set of all interpretations that satisfy it, where satisfaction. Formulas in fuzzy logic have exactly the same syntax as those of classical logic. However, they differ from classical logic in the following ways:

- An interpretation of a fuzzy language is a function, I, that maps ground atoms in the language to real numbers in the unit interval [0, 11].
- The notion of satisfaction is fuzzy-if Sat(F) denotes the set of interpretations that satisfy F, then each interpretation I of the language has a degree of membership in Sat(F).

Therefore, strictly speaking, given any formula F, and an interpretation I, we should use the notation XSat(F)(I) to denote the degree of membership of I in Sat(F). However, in order to simplify notation, we will merely write I(F) for this quantity.

Suppose I is an interpretation. If X is any set of real numbers between 0 and 1 inclusive, we will use the notation inf(X) (pronounced infimum of X) to denote the largest real number that is smaller than all elements of X. The notation sup(X) (pronounced supremum of X) denotes the smallest real number that is greater than or equal to all elements of X. Then I(F) may be defined inductively as follows:

$$
\begin{aligned}
I(\neg A) &= 1 - I(A) \\
I(A \wedge B) &= \min(I(A), I(B)) \\
I(A \vee B) &= \max(I(A), I(B)) \\
I(\forall x.F) &= \inf\{I(F[x/a]) \mid a \text{ is a ground term}\} \\
I(\exists x.F) &= \sup\{I(F[x/a]) \mid a \text{ is a ground term}\}
\end{aligned}
$$

5.5 Fuzzy Sets

We are all familiar with standard set theory (usually called naive set theory). Given a set S, we may associate with S a characteristic function Xs, defined as

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

Thus, all elements x are either in the set S or not.

In contrast to the above behavior, in fuzzy sets, the function χ_S may assign any real number in the unit interval $[0, 1]$ to element x. Thus, a fuzzy set S has an associated characteristic function, χ_S, that assigns *grades* or *levels* of membership to elements x. Intuitively, if $\chi_S(x) = 0$, then this means that x is definitely not in set S; if $\chi_S(x) = 1$, then this means that x is definitely in set S and if $\chi_S(x_1) = 0.3$ while $\chi_S(x_2) = 0.4$, then the degree of membership of x_1 in S is somewhat less than the degree of membership of x_2 in S.

In standard set theory, the concepts of union, intersection, and difference are defined in the standard ways:

$$S_1 \cup S_2 = \{x \mid x \in S_1 \text{ or } x \in S_2\}$$
$$S_1 \cap S_2 = \{x \mid x \in S_1 \text{ and } x \in S_2\}$$
$$\overline{S} = \{x \mid x \notin S\}$$

5.6 Uncertainty in Relational Databases

The relational model of data may be extended to incorporate uncertainty either at the tuple level or at the attribute level. In the tuple-level approach, we extend each tuple to have one or more uncertainty attributes. Typically, the uncertainty attribute would either be "a single real number r **G** [0, 1] or an interval [ri, r 2] of real numbers, or "a lattice element - drawn from the complete lattice of truth values being considered. For example, the first approach would perhaps be preferred when fuzzy logic is the selected mode of uncertainty. On the other hand, as we have already seen, knowing the probabilities of events only allows us, in general, to infer a probability range for conjunctions; hence, you would expect that operations such as join and Cartesian product (both akin to conjunction), and union (similar to disjunction), would only allow us to infer the existence of probability ranges, rather than point probabilities, unless there is reason to believe that something like the independence assumption may be made.

5.7 Lattice based relational databases

Suppose $(L, \sqsubseteq)$) is a complete lattice of truth values. Suppose R is a relation over schema (A_1,..., An). The tuple-based lattice extension, R^l, of relation R is a relation over schema (A_1 ,... , A_n, Unc) where dom(Unc) = L. A_1 ,... , A_n, are called the data attributes of R. Notice that R^l handles uncertainty at the tuple level, not the

attribute level. If, for example, L = [0, 1],then the following table shows a tuple-level table that extends the face table

File	Person	LLx	LLy	URx	URy	Unc
im1.gif	John Smith	10	10	20	20	0.3
im1.gif	Mark Bloom	10	10	20	20	0.6
im1.gif	Mark Bloom	30	10	40	20	0.2
im1.gif	Ted Lewis	30	10	40	20	0.8
im2.gif	Mark Bloom	50	10	60	20	1
im2.gif	Ted Lewis	10	10	20	20	1
im3.gif	Lynn Bloom	10	10	20	20	0.4
im3.gif	Elsa Bloom	10	10	20	20	0.5

Selection

Suppose C is any selection condition (defined in the standard way) on relation R^ℓ. Then the selection operator $\sigma^\ell_C(R^\ell)$ is defined as

$$\sigma^\ell_C(R^\ell) = \{(a_1, \ldots, a_n, \mu(t, R^\ell)) \mid t = (a_1, \ldots, a_n) \in \textbf{Data\_}R^\ell{}_1 \text{ where } R^\ell{}_1 = \sigma_C(R^\ell)\}$$

In other words, whenever a selection operation is performed, it is implemented as follows:

1. *Naive select:* First perform a standard select (σ) by treating the relation R^ℓ as a standard relation, and identify all tuples that satisfy the selection condition.
2. *Grouping:* Group together all data-identical tuples.
3. *LUBing:* Examine each group of data-identical tuples. Suppose that $\{t_1, \ldots, t_r\}$ are data identical. Thus, each t_i is of the form $(a_1, \ldots, a_n, \mu_i)$. Return the tuple $t = (a_1, \ldots, a_n, \mu)$, where $\mu = \mu_1 \sqcup \cdots \sqcup \mu_r$. Do this for each group.

Projection

Let π denote the standard projection operation on relational databases. Then projection in our lattice-based setting may be defined as

$$\pi^\ell_{B_1,\ldots,B_r}(R^\ell) = \{(a_1, \ldots, a_r, \mu(t, R^\ell{}_1)) \mid t = (a_1, \ldots, a_n) \in \textbf{Data\_}R^\ell{}_1 \text{ where } R^\ell{}_1 = \pi_{B_1,\ldots,B_r}(R^\ell)\}$$

The projection operator may also be expressed in the equivalent form:

$$\pi^\ell_{B_1,\ldots,B_r}(R^\ell) = \sigma^\ell_{\text{true}}\left(\pi_{B_1,\ldots,B_r,Unc}(R^\ell)\right)$$