ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)

Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.

Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.

A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.

Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Computer Science and Engineering

Academic Year 2023-24

II. B.Tech I Semster

Database Management Systems

(Common to CSE,CIC,AIDS,AIML,CSE(DS))

(20APC0502/20APC3602/20APC3002/20APC3302/20APC3201)

Prepared By

Mrs K Uthkala
Assistant Professor
Department of CSE, AITS

UNIT-1

Introduction, Introduction to Relational Model

Introduction: Database systems applications, Purpose of Database Systems, view of Data, Database Languages, Relational Databases, Database Design, Data Storage and Querying, Transaction Management, Database Architecture, Data Mining and Information Retrieval, Specialty Databases, Database users and Administrators, Introduction to Relational Model: Structure of Relational Databases, Database Schema, Keys, Schema Diagrams, Relational Query Languages, Relational Operations.

Introduction

A database-management system (DBMS) is a collection of interrelated data and a set of programs to access that data. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

Database systems are designed to manage large bodies of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access.

Database-System Applications

Databases are widely used. Here are some representative applications:

- **Enterprise Information**
  - ***Sales***: For customer, product, and purchase information.
  - ***Accounting***: For payments, receipts, account balances, assets and other accounting information.
  - ***Human resources***: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
  - ***Manufacturing***: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
  - ***Online retailers***: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.
- **Banking and Finance**
  - ***Banking***: For customer information, accounts, loans, and banking transactions.

◦ ***Credit card transactions***: For purchases on credit cards and generation of monthly statements.

◦ ***Finance***: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also, for storing real-time market data to enable online trading by customers and automated trading by the firm.

• **Universities:** For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).

• **Airlines**: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.

• **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

The file-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

• **Data redundancy and inconsistency**: the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree.

• **Difficulty in accessing data:** The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

• **Data isolation**: Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

• **Integrity problems:** The data values stored in the database must satisfy certain types of consistency constraints. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs.

• **Atomicity problems:** A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.

• **Concurrent-access anomalies:** For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously.

• **Security problems:** Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction:

For the system to be usable, it must retrieve data efficiently. Developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

• **Physical level**. The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level**. The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may

involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level**. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database.

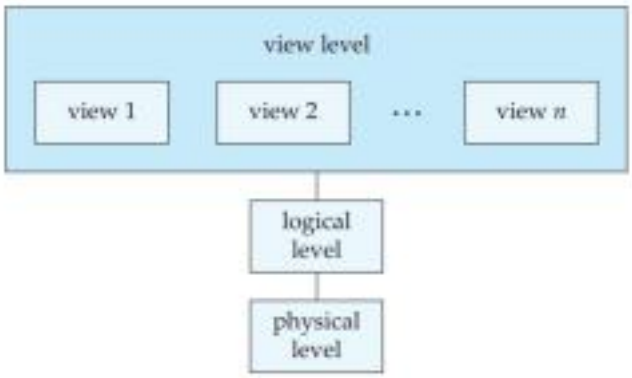


Figure 1.1 The three levels of data abstraction.

For example, we may describe a record as follows:

```
type instructor = record
        ID : char (5);
        name : char (20);
        dept_name : char (20);
        salary : numeric (8,2);
    end;
```

This code defines a new record type called instructor with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

• department, with fields dept_name, building, and budget

• course, with fields course_id, title, dept_name, and credits

• student, with fields ID, name, dept_name, and tot_cred

At the physical level, an instructor, department, or student record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database.

Instances and Schemas:

The collection of information stored in the database at a particular moment is called an instance of the database. The overall design of the database is called the database schema. Schemas are changed infrequently, if at all.

The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called subschemas, that describe different views of the database. Application programs are said to exhibit physical data independence if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Data Models:

The data model: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

• **Relational Model:** The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as relations. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

• **Entity-Relationship Model:** The entity-relationship (E-R) data model uses a collection of basic objects, called entities, and relationships among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

• **Object-Based Data Model**: Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

• **Semistructured Data Model:** The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The Extensible Markup Language (XML) is widely used to represent semistructured data.

Historically, the network data model and the hierarchical data model pre ceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data.

Database Languages

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

• Retrieval of information stored in the database

• Insertion of new information into the database

• Deletion of information from the database

• Modification of information stored in the database

There are basically two types:

• Procedural DMLs require a user to specify what data are needed and how to get those data.

• Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect, it is common practice to use the terms query language and data-manipulation language synonymously.

Data-Definition Language:

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL). We specify the storage structure and

access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language.

The data values stored in the database must satisfy certain **consistency constraints**. For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated.

Database systems implement integrity constraints that can be tested with minimal overhead:

- **Domain Constraints**: A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

- **Referential Integrity.** There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the dept name value in a course record must appear in the dept name attribute of some record of the department relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

- **Assertions:** An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, “Every department must have at least five courses offered every semester” must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

- **Authorization:** We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of authorization, the most common being: read authorization, which allows reading, but not modification, of data; insert authorization, which allows insertion of new data, but not modification of existing

data; update authorization, which allows modification, but not deletion, of data; and delete authorization, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**, which contains **metadata**— that is, data about data. The data dictionary is considered to be a special type of table that can only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

Relational Databases

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationships among those data.

Table:

Each table has multiple columns and each column has a unique name.

The first table, the instructor table, shows, for example, that an instructor named Einstein with ID 22222 is a member of the Physics department and has an annual salary of $95,000. The

second table, department, shows, for example, that the Biology department is located in the Watson building and has a budget of $90,000. Of course, a real-world university would have many more departments and instructors.

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

dept_name	building	budget
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Figure 1.2 A sample relational database.

The relational model is an example of **a record-based model**. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

Data-Manipulation Language:

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQL query that finds the names of all instructors in the History department:

SQL> select instructor.name from instructor where instructor.dept_name = 'History';

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved, and the name attribute of these rows must be displayed.

Queries may involve information from more than one table. For instance, the following query finds the instructor ID and department name of all instructors associated with a department with budget of greater than $95,000.

SQL> select instructor.ID, department.dept_name from instructor, department where instructor.dept_name= department.dept_name and department.budget > 95000;

Data-Definition Language:

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc. For instance, the following SQL DDL statement defines the department table:

SQL> create table department (dept_name char (20), building char (15), budget numeric (12,2)); **Database Access from Application Programs:**

There are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a host language, such as C, C++, or Java, with embedded SQL queries that access the data in the database. **Application programs** are programs that are used to interact with the database in this fashion.

To access the database, DML statements need to be executed from the host language. There are two ways to do this:

• By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results.

The Open Database Connectivity (ODBC) standard for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

• By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the **DML precompiler**, converts the DML statements to normal procedure calls in the host language.

Database Design

Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues.

Design Process:

The **initial phase** of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of **user requirements**.

Next, the designer chooses a data model, and by applying the concepts of the

chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. The designer can also examine the design to remove any redundant features.

A fully developed conceptual schema indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or

I

transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

The process of moving from an abstract data model to the implementation of the database proceeds in **two** final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

Database Design for a University Organization:

• The university is organized into departments. Each department is identified by a unique name (dept name), is located in a particular building, and has a budget.

• Each department has a list of courses it offers. Each course has associated with it a course id, title, dept name, and credits, and may also have associated prerequisites.

• Instructors are identified by their unique ID. Each instructor has name, associated department (dept name), and salary.

• Students are identified by their unique ID. Each student has a name, an associated major department (dept name), and tot cred (total credit hours the student earned thus far).

• The university maintains a list of classrooms, specifying the name of the building, room number, and room capacity.

• The university maintains a list of all classes (sections) taught. Each section is identified by a course id, sec id, year, and semester, and has associated with it a semester, year, building, room number, and time slot id (the time slot when the class meets).

• The department has a list of teaching assignments specifying, for each instructor, the sections the instructor is teaching.

• The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).

The Entity-Relationship Model:

The **entity-relationship (E-R)** data model uses a collection of basic objects, called entities, and relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes dept_name, building, and budget may describe one particular department in a university, and they form attributes of the department entity set. Similarly, attributes ID, name, and salary may describe an instructor entity.

A **relationship** is an association among several entities. For example, a member relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively. The overall logical structure (schema) of a database can be expressed graphically by an entity
relationship (E-R) diagram. There are several ways in which to draw these diagrams. One of the most

popular is to use the **Unified Modeling Language (UML).** In the notation we use, which is based on UML, an E-R diagram is represented as follows:

Figure 1.3 A sample E-R diagram.

• Entity sets are represented by a rectangular box with the entity set name in the header and the attributes listed below it.

• Relationship sets are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set.

Normalization:

The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve

information easily. The approach is to design schemas that are in an appropriate **normal form**. The most common approach is to use **functional dependencies.**

Among the undesirable properties that a bad design may have are:

• Repetition of information

• Inability to represent certain information

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 1.4 The *faculty* table.

Notice that there are two rows in faculty that contain repeated information about the History department, specifically, that department's building and budget. The repetition of information in our alternative design is undesirable.

The **null** value indicates that the value does not exist (or is not known). An unknown value may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists).

Data Storage and Querying,

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the -

- storage manager and
- the query processor components.

The storage manager:

- The storage manager is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible for the interaction with the file manager.
- The storage manager translates the various DML statements into low-level file-system commands.
- The storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:-

• **Authorization and integrity manager** - it tests for the satisfaction of integrity constraints and checks the authority of users to access data.

• **Transaction manager** - it ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

• **File manager**- it manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

• **Buffer manager-** it is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the phys ical system implementation:-

• **Data files**, which store the database itself.

• **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.

• **Indices,** which can provide fast access to data items. Like the index in this textbook, a database index provides pointers to those data items that hold a particular value.

The Query Processor:

- The query processor is important because it helps the database system to simplify and facilitate access to data.
- The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level.

The query processor components include:-

- **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
- **Query evaluation engine,** which executes low-level instructions generated by the DML compiler.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs ***query optimization***; that is, it picks the lowest cost evaluation plan from among the alternatives.

Transaction Management

A **transaction** is a collection of operations that performs a single logical function in a database application.
Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database consistency constraints.

-Atomicity : all-or-none requirement.

Ex: a funds transfer, as in which one department account (say A) is debited and another department account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur.

- Consistency : requirement of correctness.

Ex: it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved.

-Durability : requirement of persistence.

Ex: Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure.

Ensuring the atomicity and durability properties is the responsibility of the database system itself—specifically, of the **recovery manager.**

-Because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database.

-The database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform **failure recovery,** that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

- **concurrency-control manager :-** to control the interaction among the concurrent transactions, to ensure the consistency of the database.

-transaction manager :- consists of the concurrency-control manager and the recovery manager.

Database Architecture

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines.
-We can therefore differentiate between **client** machines, on which remote

database users work, and **server** machines, on which the database system runs.
-Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

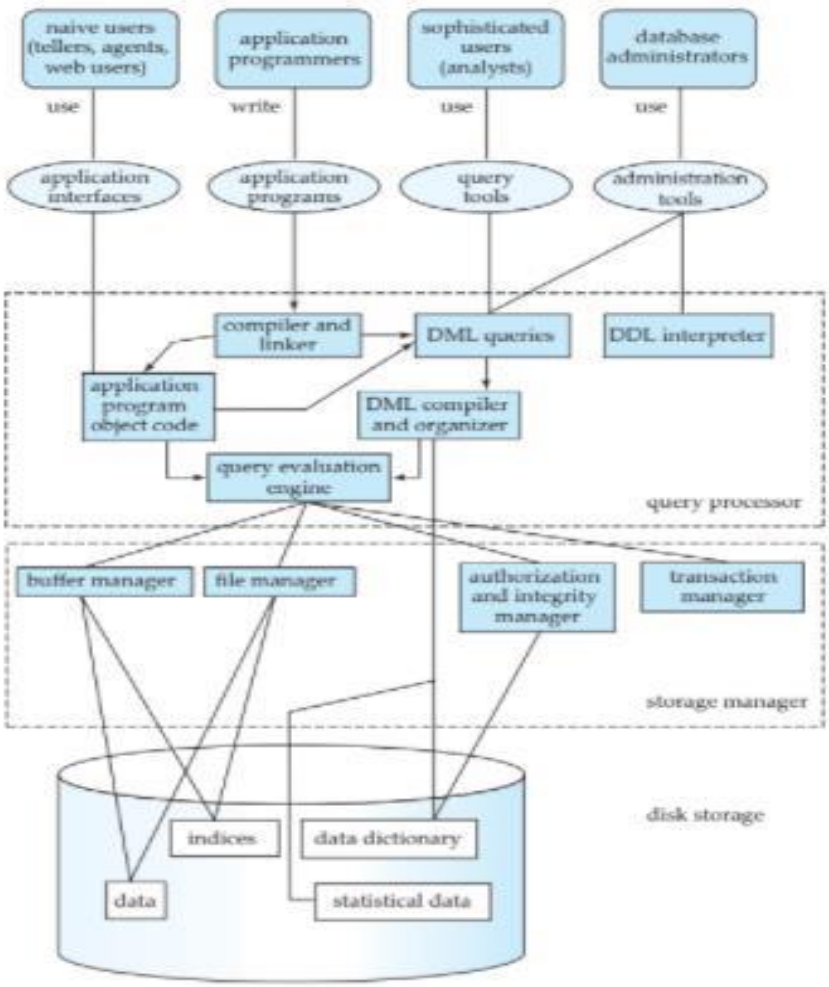


Figure 1.5 System structure.

-Database applications are usually partitioned into two or three parts, as in Figure 1.6. In a **two-tier architecture**, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements.
-In contrast, in a **three-tier architecture,** the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client communicates with an **application server,** usually through a forms interface.The application server in turn communicates with a database system to access data.
-Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

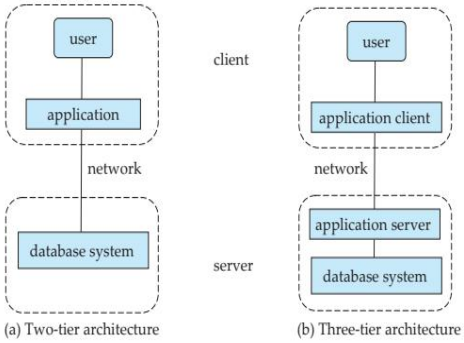


Figure 1.6 Two-tier and three-tier architectures.

Data Mining and Information Retrieval

The term **data mining** refers loosely to the process of semi automatically analyzing large databases to find useful patterns.

Ex: Like knowledge discovery in artificial intelligence (also called machine learning) or statistical analysis, data mining attempts to discover rules and patterns from data.

"data mining deals with "knowledge discovery in databases."

-Some types of knowledge discovered from a database can be represented by a set of **rules**.

Ex: The following is an example of a rule, stated informally: "Young women with annual incomes greater than $50,000 are the most likely people to buy small sports cars." Of course such rules are not universally true, but rather have degrees of "support" and "confidence."

-Usually there is a manual component to data mining, consisting of preprocessing data to a form acceptable to the algorithms, and postprocessing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, and manual interaction may be needed to pick useful types of patterns.

Specialty Databases

Several application areas for database systems are limited by the restrictions of the relational data model. As a result, researchers have developed several data models to deal with these application domains, including object-based data models and semistructured data models.

- **Object-Based Data Models:** Object-oriented programming has become the dominant software-development methodology. This led to the development of an **object-oriented data model** that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Inheritance, object identity, and encapsulation (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modeling.
- **Semistructured Data Models :** Semistructured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes.

  -The XML language was initially designed as a way of adding markup information to text documents.XML provides a way to represent data that have nested structure, and furthermore allows a great deal of flexibility in structuring of data, which is important for certain kinds of nontraditional data.

Database users and Administrators

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

Database Users:

There are four different types of database-system users, differentiated by the way they expect to interact with the system.

Naive users:

These users are also called Unsophisticated Users. They don't have any type of knowledge with the system. These users interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account A to account B invokes a program called transfer. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

Application programmers:

These users are computer professionals who write application programs. Application Programmers can choose from many tools to develop user interfaces. Rapid application development (RAD) tools are tools that enable an

application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language.

Sophisticated users:

These users are interacting with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a query processor, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category. Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways.

Specialized users:

These users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems. And these users write specialized application program for queries, which cannot be answered directly.

Database Administrator:

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a database administrator (DBA). The functions of a DBA include:

• **Schema definition**: The DBA creates the original database schema by executing a set of data definition statements in the DDL.

• **Storage structure and access-method definition.**

• **Schema and physical-organization modification:** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

• **Granting of authorization for data access:** By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone

attempts to access the data in the system.

• **Routine maintenance:** Examples of the database administrator's routine maintenance activities are:

-Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.

-Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

-Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

Introduction to Relational Model

Structure of Relational Databases

A relational database consists of a collection of **tables,** each of which is assigned a unique name. For example, consider the instructortable of Figure 2.1, which stores information about instructors. The table has four column headers: ID, name, dept name, and salary. Each row of this table records information about an instructor, consisting of the instructor's ID, name, dept name, and salary.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.

In general, a row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.

- **Tuple :** a tuple is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an **n-tuple** of values, i.e., a tuple with n values, which corresponds to a row in a table.
- **Relation :** in the relational model the term **relation** is used to refer to a

table, while the term tuple is used to refer to a row.

- **Attribute :** the term **attribute** refers to a column of a table.

  Ex: Figure 2.1, we can see that the relation instructor has four attributes: ID, name, dept name, and salary.
- **relation instance :** the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows.
  Ex: The instance of the instructor shown in Figure 2.1 has 12 tuples, corresponding to 12 instructors.
  -For each attribute of a relation, there is a set of permitted values, called the
  domain of that attribute. A domain is atomic if elements of the domain are considered to be indivisible units. For example, suppose the table instructor had an attribute phone number,
  which can store a set of phone numbers corresponding to the instructor. Then the
  domain of phone number would not be atomic, since an element of the domain is a
  set of phone numbers.
- **null value :** The null value is a special value that signifies that the value is unknown or
  does not exist.

Database Schema

Database schema: it is the logical design of the database.

Database instance: it is a snapshot of the data in the database at a given instant in time.

-the concept of a **relation schema** corresponds to the programming-language notion of type definition. A relation schema consists of a list of attributes and their corresponding domains.

-The concept of a **relation instance** corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time.

Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as instructor, to refer to both the schema and the instance.

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

Consider the department relation of Figure 2.5. The schema for that relation is

department (dept name, building, budget)

Note that the attribute dept name appears in both the instructor schema and the department schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

-Let us continue with our university database example. Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class. The schema is

section (course id, sec id, semester, year, building, room number, time slot id)

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.6 The *section* relation.

Figure 2.6 shows a sample instance of the section relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

teaches (ID, course id, sec id, semester, year)

Keys

A key is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key, and then we select any of them as the Primary Key. So for each entity set we choose a key. And the set of attributes contains a key. The key attribute is represented with underline. Each attribute in the primary key is underlined.

The values of the attribute values of a tuple must be such that they can uniquely identify the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

- A **superkey** is a set of one or more attributes that, taken collectively, allow us to uniquely identify a tuple in the relation.
- It is possible that several distinct sets of attributes could serve as a **candidate key.**
- The term primary key to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time.
- A relation, say r1, may include among its attributes the primary key of another relation, say r2.

This attribute is called a foreign key from r1, referencing r2. The relation r1 is also called the referencing relation of the foreign key dependency, and r2 is called the referenced relation of theforeign key.

Schema Diagrams

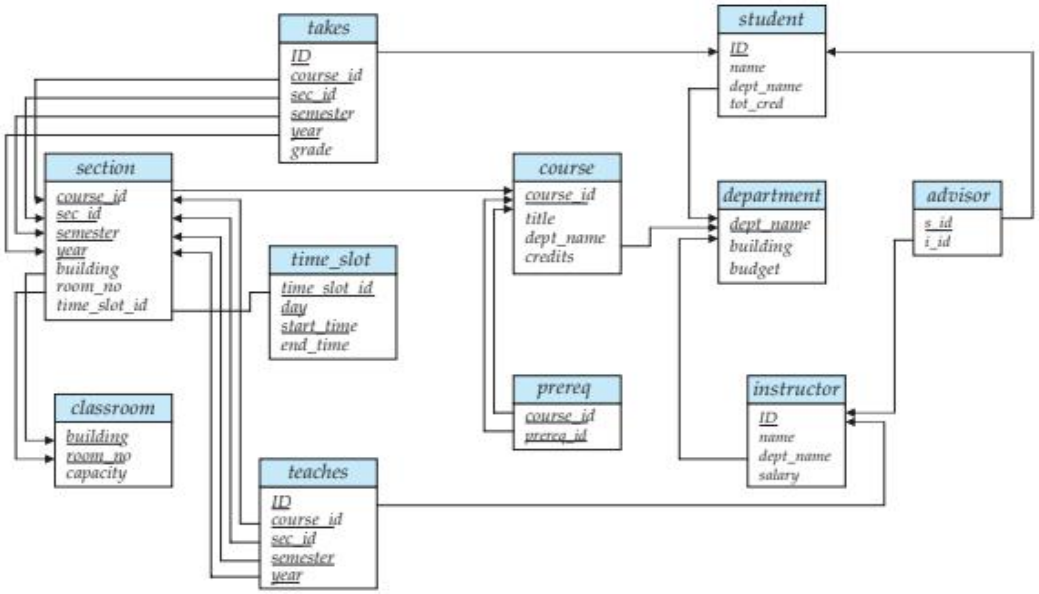


Figure 2.8 Schema diagram for the university database.

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams.** Figure 2.8 shows the schema diagram for our university organization.

- Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box.

- Primary key attributes are shown underlined.
- Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

Relational Query Languages

A **query language** is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as

- **procedural :** In a **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result.

-nonprocedural : In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information

Query languages used in practice include elements of both the procedural and the nonprocedural approaches.

- The **join** operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple.
- a **natural join**, a tuple from the instructor relation matches a tuple in the department relation if the values of their dept name attributes are the same.

ID	*name*	*salary*	*dept_name*	*building*	*budget*
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Figure 2.12 Result of natural join of the *instructor* and *department* relations.

Figure 2.12 shows an example
of joining the tuples from the instructor and department tables with the new tuples
showing the information about each instructor and the department in which she
is working. This result was formed by combining each tuple in the instructor
relation with the tuple in the department relation for the instructor's department.

- The **Cartesian product** operation combines tuples from two relations, but unlike the join operation, its result contains all pairs of tuples from the two relations, regardless of whether their attribute values match.
- The **union** operation performs a set union of two "similarly structured" tables (say a table of all graduate students and a table of all undergraduate students). For example, one can obtain the set of all students in a department. Other set operations, such as **intersection** and **set difference** can be performed as well.

ID	*name*	*dept_name*	*salary*
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Figure 2.10 Result of query selecting *instructor* tuples with salary greater than $85000.

The most frequent operation is the selection of specific tuples from a single relation (say instructor) that satisfies some particular predicate (say salary

>$85,000). The result is a new relation that is a subset of the original relation (instructor). For example, if we select tuples from the instructor relation of Figure 2.1,satisfying the predicate “salary is greater than $85000”, we get the result shown in Figure 2.10.

RELATIONAL ALGEBRA

The relational algebra defines a set of operations on relations, paralleling the usual algebraic operations such as addition, subtraction or multiplication, which operate on numbers. Just as algebraic operations on numbers take one or more numbers as input and return a number as output, the relational algebra operations typically take one or two relations as input and return a relation as output.

Relational algebra is covered in detail in Chapter 6, but we outline a few of the operations below.

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{salary>=85000}(instructor)$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID,salary}(instructor)$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
$\bowtie$ (Natural join)	$instructor \bowtie department$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
$\times$ (Cartesian product)	$instructor \times department$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
$\cup$ (Union)	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$ Output the union of tuples from the two input relations.

The relational algebra provides a set of operations that take one or more relations as input and return a relation as an output. Practical query languages such as SQL are based on the relational algebra, but add a number of useful syntactic features.

UNIT- 2

Introduction to SQL, Advanced SQL

Introduction to SQL: Overview of the SQL Query Language, SQL Data Definition, Basic Structure of SQL Queries, Additional Basic Operations, Set Operations, Null Values, Aggregate Functions, Nested Sub-queries, Modification of the Database.

Intermediate SQL: Joint Expressions, Views, Transactions, Integrity Constraints, SQL Data types and schemas, Authorization.

Advanced SQL: Accessing SQL from a Programming Language, Functions and Procedures, Triggers, Recursive Queries, OLAP, Formal relational query languages.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.2 The *course* relation.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.6 The *section* relation.

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 2.7 The *teaches* relation.

Overview of the SQL Query Language

IBM developed the original version of SQL, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language).

In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, followed by SQL:1999, SQL:2003, SQL:2006, and most recently SQL:2008.

The SQL language has several parts:

- **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language (DML)**. The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition.** The SQL DDL includes commands for defining views.
- **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

SQL Data Definition

The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Basic Types:

The SQL standard supports a variety of built-in types, including:

- **char(n):** A fixed-length character string with user-specified length n. The full form, character, can be used instead.
- **varchar(n):** A variable-length character string with user-specified maximum length n. The full form, character varying, is equivalent.
- **int:** An integer (a finite subset of the integers that is machine dependent). The full form, integer, is equivalent.
- **smallint:** A small integer (a machine-dependent subset of the integer type).
- **numeric(p, d):** A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.

• **real, double precision:** Floating-point and double-precision floating-point numbers with machine-dependent precision.
• **float(n):** A floating-point number, with precision of at least n digits.

Basic Schema Definition:

create table command:

The general form of the create table command is:

SQL>create table r (A1 D1, A2 D2, ..., An Dn, integrity-constraint1, ..., integrity-constraintk);

Where r is the name of the relation, each Ai is the name of an attribute in the schema of relation r, and Di is the domain of attribute Ai ; that is, Di specifies the type of attribute Ai along with optional constraints that restrict the set of allowed values for Ai .

The semicolon shown at the end of the create table statements, as well as at the end of other SQL statements.

The following command creates a relation department in the database.

SQL>create table department (dept_name varchar (20), building varchar (15), budget numeric (12,2), primary key (dept_name));

SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

• **primary key (Aj1 , Aj2 ,..., Ajm):** The primary-key specification says that attributes Aj1 , Aj2 ,..., Ajm form the primary key for the relation. The primary key attributes are required to be nonnull and unique; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

Example:

SQL>create table department (dept_name varchar (20), building varchar (15), budget numeric (12,2), primary key (dept)_name));

• **foreign key (Ak1 , Ak2 ,..., Akn)references s:** The foreign key specification says that the values of attributes (Ak1 , Ak2 ,..., Akn) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.

Example:

SQL>create table course (course_id varchar (7), title varchar (50), dept_name varchar (20), credits numeric (2,0), primary key (course_id), foreign key (dept_name) references department);

• **not null:** The not null constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute.

Example:

SQL>create table instructor (ID varchar (5), name varchar (20) not null, dept_name varchar (20), salary numeric (8,2), primary key (ID), foreign key (dept_name) references department);

Other examples: SQL data definition for part of the university database.

SQL>create table section (course_id varchar (8), sec_id varchar (8), semester varchar (6), year numeric (4,0), building varchar (15), room_number varchar (7), time_slot_id varchar (4), primary key (course_id, sec_id, semester, year), foreign key (course_id) references course);

SQL>create table teaches (ID varchar (5), course_id varchar (8), sec_id varchar (8), semester varchar (6), year numeric (4,0), primary key (ID, course_id, sec_id, semester, year), foreign key (course_id, sec_id, semester, year) references section, foreign key (ID) references instructor);

A newly created relation is empty initially. We can use **the insert command** to load data into the relation.

SQL>insert into instructor values (10211, 'Smith', 'Biology', 66000);

We can use **the delete command** to delete tuples from a relation. The command

SQL>delete from student;

The **drop table command** deletes all information about the dropped relation from the database. The command

SQL>drop table r;

It deletes not only all tuples of r, but also the schema for r.

SQL>delete from r;

It deletes all tuples in r.

We use the **alter table** command to add attributes to an existing relation.

alter table r add A D;

where r is the name of an existing relation, A is the name of the attribute to be added, and D is the type of the added attribute. We can drop attributes from a relation by the command

alter table r drop A;

Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses:

- select,
- from, and
- where

The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and select clauses, and then produces a relation as the result.

The role of each clause is as follows:

- The **select clause** is used to list the attributes desired in the result of a query.
- The **from clause** is a list of the relations to be accessed in the evaluation of the query.
- The **where clause** is a predicate involving attributes of the relation in the from clause.

A typical SQL query has the form

select A1, A2,..., An from r1, r2,...,rm where P;

Each Ai represents an attribute, and each ri a relation. P is a predicate. If the where clause is omitted, the predicate P is **true**.

Queries on a Single Relation:

Let us consider a simple query using our university example,

Q: "Find the names of all instructors."

SQL>select name from instructor;

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of "**select** *name* **from** *instructor*".

Instructor names are found in the instructor relation, so we put that relation in the **from** clause. The instructor's name appears in the **name** attribute, so we put that in the **select** clause.

Q: "Find the department names of all instructors."
SQL>select dept_name from instructor;

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of "**select** *dept_name* **from** *instructor*".

In those cases where we want to force the **elimination of duplicates**, we insert the keyword **distinct** after **select**.
SQL>select distinct dept_name from instructor;

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:
SQL>select all dept_name from instructor;
Since duplicate retention is the default, we shall not use all in our examples.

The **select clause** may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query:
SQL>select ID, name, dept_name, salary * 1.1 from instructor;

Q: "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."
SQL>select name from instructor where dept_name = 'Comp. Sci.' and salary > 70000;

name
Katz
Brandt

Figure 3.4 Result of "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."

SQL allows the use of the logical connectives **and, or,** and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators **<=, >, >=, =,** and **<>**.

Queries on Multiple Relations:

Queries often need to access information from multiple relations. An an example, suppose we want to answer the query **"Retrieve the names of all instructors, along with their department names and department building name."**

Looking at the schema of the relation **instructor**, we realize that we can get the department name from the attribute **dept_name**, but the department building name is present in the attribute **building** of the relation **department**. To answer the query, each tuple in the **instructor relation** must be matched with the tuple in the **department relation** whose **dept_name** value matches the dept_name value of the instructor tuple.

The above query can be written in SQL as

SQL>select name, instructor.dept_name, building from instructor, department where instructor.dept_name= department.dept_name;

name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.5 The result of "Retrieve the names of all instructors, along with their department names and department building name."

Note that the attribute dept_name occurs in both the relations instructor and department, and the relation name is used as a prefix (in **instructor.dept_name**, and **department.dept_name**) to make clear to which attribute we are referring.

The **from clause** by itself defines a **Cartesian product** of the relations listed in the clause. It is defined formally in terms of set theory, but is perhaps best understood as an iterative process that generates tuples for the result relation of the from clause.

```
for each tuple t1 in relation r1
        for each tuple t2 in relation r2
                ...
                for each tuple tm in relation rm
                        Concatenate t1, t2,..., tm into a single tuple t
                        Add t into the result relation
```

For example, the relation schema for the Cartesian product of relations instructor and teaches is:

(instructor.ID, instructor.name, instructor.dept_name, instructor.salary teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

We can then write the relation schema as:

(instructor.ID, name, dept_name, salary teaches.ID, course_id, sec_id, semester, year)

To illustrate, consider the instructor relation in Figure 2.1 and the teaches relation in Figure 2.7. Their Cartesian product is shown in Figure 3.6, which includes only a portion of the tuples that make up the Cartesian product result.

inst_ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

The Cartesian product by itself combines tuples from instructor and teaches that are unrelated to each other. Each tuple in instructor is combined with every tuple in teaches, even those that refer to a different instructor. The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.

We would expect a query involving instructor and teaches to combine a particular tuple t in instructor with only those tuples in teaches that refer to the same instructor to which t refers. That is, we wish only to match teaches tuples with instructor tuples that have the same ID value. The following SQL query ensures this condition, and outputs the instructor name and course identifiers from such matching tuples.

SQL>select name, course_id from instructor, teaches where instructor.ID= teaches.ID;

If we only wished to find instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the where clause, as shown below.

SQL>select name, course_id from instructor, teaches where instructor.ID= teaches.ID and instructor.dept_name = 'Comp. Sci.';

name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.7 Result of "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

1. Generate a Cartesian product of the relations listed in the from clause
2. Apply the predicates specified in the where clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the select clause.

The above sequence of steps helps make clear what the result of an SQL query should be, not how it should be executed.

The Natural Join:

The natural join operation operates on two relations and produces a relation as the result. Consider the query **"For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught"**, which we wrote earlier as:

SQL>select name, course_id from instructor, teaches where instructor.ID= teaches.ID;

This query can be written more concisely using the **natural-join** operation in SQL as:
SQL>select name, course_id from instructor natural join teaches;
Both of the above queries generate the same result.

A **from** clause in an SQL query can have multiple relations combined using natural join, as shown here:

select A1, A2,..., An from r1 natural join r2 natural join ... natural join rm where P;

More generally, a from clause can be of the form
from E1, E2,..., En

where each Ei can be a single relation or an expression involving natural joins. For example, suppose we wish to answer the query **"List the names of instructors along with the the titles of courses that they teach."** The query can be written in SQL as follows:

SQL>select name, title from instructor natural join teaches, course where teaches.course_id= course.course_id;

Additional Basic Operations

There are number of additional basic operations that are supported in SQL.

The Rename Operation:

SQL provides a way of renaming the attributes of a result relation. It uses the **as clause**, taking the form:

old-name as new-name

The **as** clause can appear in both the **select** and **from** clauses

For example, if we want the attribute ***name*** name to be replaced with the name ***instructor_name***, we can rewrite the preceding query as:

SQL>select name as instructor_name, course_id from instructor, teaches where instructor.ID= teaches.ID;

To illustrate, we rewrite the query **"For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."**

SQL>select T.name, S.course_id from instructor as T, teaches as S where T.ID= S.ID;

Q: **"Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."**

SQL>select distinct T.name from instructor as T, instructor as S where T.salary > S.salary and S.dept_name = 'Biology';

String Operations:

SQL specifies strings by enclosing them in single quotes, for example, 'Computer'.

Pattern matching can be performed on strings, using the operator **like**.

We describe patterns by using two special characters:

- **Percent (%):** The % character matches any substring.
- **Underscore (_):** The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
- ' ' matches any string of exactly three characters.
- ' %' matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator.

Q: **"Find the names of all departments whose building name includes the substring 'Watson'."**

SQL>select dept_name from department where building like '%Watson%';

We define the **escape character** for a like comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a backslash **(\)** as the escape character:

- like 'ab\%cd%' escape '\' matches all strings beginning with "ab%cd".
- like 'ab\\cd%' escape '\' matches all strings beginning with "ab\cd".

Attribute Specification in Select Clause:

The **asterisk** symbol **" * "** can be used in the select clause to denote "**all attributes**."
Thus, the use of **instructor.*** in the select clause of the query:

SQL>select instructor.* from instructor, teaches where instructor.ID= teaches.ID;

indicates that all attributes of instructor are to be selected. A select clause of the form **select *** indicates that all attributes of the result relation of the **from** clause are selected.

Ordering the Display of Tuples:

The **order by** clause causes the tuples in the result of a query to appear in sorted order. **To list in alphabetic order all instructors in the Physics department**, we write:

SQL>select name from instructor where dept_name = 'Physics' order by name;

By default, the **order by** clause lists items in **ascending** order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order.

Suppose that we wish to list the entire instructor relation in descending order of salary.

SQL>select * from instructor order by salary desc, name asc;

Where Clause Predicates:

SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to **find the names of instructors with salary amounts between $90,000 and $100,000**, we can use the between comparison to write:

SQL>select name from instructor where salary between 90000 and 100000;

instead of:

SQL>select name from instructor where salary <= 100000 and salary >= 90000;

Similarly, we can use the **not between** comparison operator.

Q:**"Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course."**
SQL>select name, course_id from instructor, teaches where instructor.ID= teaches.ID and dept_name = 'Biology';

Q: **"Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course."**

We show below the modified form of the SQL query that does not use natural join.
SQL>select name, course_id from instructor, teaches where instructor.ID= teaches.ID and dept_name = 'Biology';

Thus, the preceding SQL query can be rewritten as follows:

SQL>select name, course_id from instructor, teaches where (instructor.ID, dept_name) = (teaches.ID, 'Biology');

Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −. We shall now construct queries involving the union, intersect, and except operations over two sets.

• The set of all courses taught in the Fall 2009 semester:

SQL>select course_id from section where semester = 'Fall' and year= 2009;

course_id
CS-101
CS-347
PHY-101

Figure 3.9 The *c1* relation, listing courses taught in Fall 2009.

• The set of all courses taught in the Spring 2010 semester:

SQL>select course_id from section where semester = 'Spring' and year= 2010;

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.10 The *c2* relation, listing courses taught in Spring 2010.

we shall refer to the relations obtained as the result of the preceding queries as c1 and c2, respectively.

The Union Operation:

To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both, we write:

SQL>**(select course_id from section where semester = 'Fall' and year= 2009) union (select course_id from section where semester = 'Spring' and year= 2010);**

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.11 The result relation for *c1* union *c2*.

The union operation automatically eliminates duplicates.

If we want to retain all duplicates, we must write **union all** in place of **union**:

SQL>(select course_id from section where semester = 'Fall' and year= 2009) union all (select course_id from section where semester = 'Spring' and year= 2010);

The Intersect Operation:

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

SQL>(select course_id from section where semester = 'Fall' and year= 2009) intersect (select course_id from section where semester = 'Spring' and year= 2010);

course_id
CS-101

Figure 3.12 The result relation for *c1* intersect *c2*.

The intersect operation automatically eliminates duplicates.

If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

SQL>(select course_id from section where semester = 'Fall' and year= 2009) intersect all (select course_id from section where semester = 'Spring' and year= 2010);

The Except Operation:

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

SQL>(select course_id from section where semester = 'Fall' and year= 2009) except (select course_id from section where semester = 'Spring' and year= 2010);

course_id
CS-347
PHY-101

Figure 3.13 The result relation for *c1* except *c2*.

The operation automatically eliminates duplicates in the inputs before performing set difference.

If we want to retain duplicates, we must write **except all** in place **of except**:

SQL>(select course_id from section where semester = 'Fall' and year= 2009) except all (select course_id from section where semester = 'Spring' and year= 2010);

Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example +, −, ∗, or /) is null if any of the input values is null. For example, if a query has an expression r.A+ 5, and r.A is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison "1 < null".

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

- **and:** The result of **true** and **unknown** is **unknown**, **false** and **unknown** is **false**, while **unknown** and **unknown** is **unknown**.
- **or:** The result of **true** or **unknown** is **true**, **false** or **unknown** is **unknown**, while **unknown** or **unknown** is **unknown**.
- **not:** The result of **not unknown** is **unknown**.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the instructor relation with null values for salary, we write:

SQL>select name from instructor where salary is null;

Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Basic Aggregation:

Q:"Find the average salary of instructors in the Computer Science department." We write this query as follows:

SQL>select avg (salary) from instructor where dept_name= 'Comp. Sci.';

we can give a meaningful name to the attribute by using the **as** clause as follows:

SQL>select avg (salary) as avg_salary from instructor where dept_name= 'Comp. Sci.';

Q:"Find the total number of instructors who teach a course in the Spring 2010 semester."

SQL>select count (distinct ID) from teaches where semester = 'Spring' and year = 2010;

We use the aggregate function **count** frequently to count the number of tuples in a relation.

Q:"find the number of tuples in the course relation", we write:

SQL>select count (*) from course;

SQL does not allow the use of **distinct** with **count (*).** It is legal to use **distinct** with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct** to specify duplicate retention, but, since **all** is the default, there is no need to do so.

Aggregation with Grouping:

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause.

The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 3.14 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

As an illustration, consider the query **"Find the average salary in each department."** We write this query as follows:

SQL>select dept_name, avg (salary) as avg_salary from instructor group by dept_name;

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.15 The result relation for the query "Find the average salary in each department".

Q: "Find the number of instructors in each department who teach a course in the Spring 2010 semester."

SQL>select *dept_name*, **count** (**distinct** *ID*) **as** *instr_count* **from** *instructor* **natural join** *teaches* **where** *semester* = 'Spring' **and** *year* = 2010 **group by** *dept_name*;

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.16 The result relation for the query "Find the number of instructors in each department who teach a course in the Spring 2010 semester."

The Having Clause:

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000.

This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used.We express this query in SQL as follows:

SQL>select *dept_name*, **avg** (*salary*) **as** *avg_salary* **from** *instructor* **group by** *dept_name*
having avg (*salary*) > 42000;

dept_name	avg(avg_salary)
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.17 The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.

2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.

3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.

5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

Q: "For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students."

SQL>select *course_id*, *semester*, *year*, *sec_id*, **avg** (*tot cred*) **from** *takes* **natural join** *student* **where** *year* = 2009 **group by** *course_id*, *semester*, *year*, *sec_id* **having count** (*ID*) >= 2;

Aggregation with Null and Boolean Values:

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the *instructor* relation have a null value for *salary*.

SQL> select sum (*salary*) **from** *instructor*;

The SQL standard says that the **sum** operator should ignore *null* values in its input.

NOTE: All aggregate functions except **count (*)** ignore null values in their input collection.

Nested Subqueries

A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the **where** clause.

Set Membership:

The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

Q: "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters."

SQL> select distinct *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and** *course_id* **in** (**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010);

We use the **not in** construct in a way similar to the **in** construct. For example, to **find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester**, we can write:

SQL> select distinct *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and**
course_id **not in** (**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010);

Q: List the names of instructors whose names are neither "Mozart" nor "Einstein".
SQL> select distinct *name* **from** *instructor* **where** *name* **not in** ('Mozart', 'Einstein');

Q:"find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011" as follows:
SQL> select count (**distinct** *ID*)
from *takes*
where (*course id*, *sec id*, *semester*, *year*) **in** (**select** *course id*, *sec id*, *semester*, *year*
from *teaches*
where *teaches.ID*= 10101);

Set Comparison:

The ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."

SQL> select distinct *T.name* **from** *instructor* **as** *T*, *instructor* **as** *S* **where** *T.salary* > *S.salary* **and** *S.dept_name* = 'Biology';

The phrase "greater than at least one" is represented in SQL by > **some**.

SQL> select *name* **from** *instructor* **where** *salary* > **some** (**select** *salary* **from** *instructor* **where** *dept_name* = 'Biology');

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons. As an exercise, verify that = **some** is identical to **in**, whereas <> **some** is *not* the same as **not in.**

Q:"find the names of all instructors that have a salary value greater than that of each instructor in the Biology department". The construct > **all** corresponds to the phrase "greater than all."

SQL> select *name* **from** *instructor* **where** *salary* > **all** (**select** *salary* **from** *instructor* **where** *dept_name* = 'Biology');

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons. As an exercise, verify that <> **all** is identical to **not in**, whereas = **all** is *not* the same as **in**.

Q: "Find the departments that have the highest average salary."
SQL> select *dept_name* **from** *instructor* **group by** *dept_name* **having avg** (*salary*) >= **all** (**select avg** (*salary*) **from** *instructor* **group by** *dept_name*);

Test for Empty Relations:

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty.

Using the **exists** construct, we can write the query **"Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"** in still another way:

SQL> select *course_id* **from** *section* **as** *S* **where** *semester* = 'Fall' **and** *year*= 2009 **and**
exists (**select** * **from** *section* **as** *T* **where** *semester* = 'Spring' **and** *year*= 2010 **and** *S.course id*= *T.course_id*);

The above query also illustrates a feature of SQL where a correlation name from an outer query (*S* in the above query), can be used in a subquery in the **where** clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct.We can use the **not exists** construct to simulate the set containment (that is, superset) operation:We canwrite "relation *A*contains relation *B*" as "**not exists** (*B* **except** *A*)."

Q: "Find all studentswho have taken all courses offered in the Biology department." Using the **except** construct:
SQL> select distinct *S.ID*, *S.name* **from** *student* **as** *S* **where not exists** ((**select** *course_id* **from** *course* **where** *dept_name* = 'Biology') **except** (**select** *T.course id* **from** *takes* **as** *T* **where** *S.ID* = *T.ID*));

Test for the Absence of Duplicate Tuples:

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct9 returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query **"Find all courses that were offered at most once in 2009"** as follows:
SQL>select *T.course_id* **from** *course* **as** *T* **where unique** (**select** *R.course_id* **from** *section* **as** *R* **where** *T.course_id*= *R.course_id* **and** *R.year* = 2009);

An equivalent version of the above query not using the **unique** construct is:
SQL> select *T.course_id* **from** *course* **as** *T* **where** 1 <= (**select count**(*R.course_id*) **from** *section* **as** *R*
where *T.course id*= *R.course_id* **and** *R.year* = 2009);

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct.

Q: "Find all courses that were offered at least twice in 2009" as follows:
SQL> select *T.course_id* **from** *course* **as** *T* **where not unique** (**select** *R.course_id* **from** *section* **as** *R*
where *T.course_id*= *R.course_id* **and** *R.year* = 2009);

Subqueries in the From Clause:

SQL allows a subquery expression to be used in the **from** clause. Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000."
SQL> select *dept_name*, *avg_salary* **from** (**select** *dept_name*, **avg** (*salary*) **as** *avg_salary* **from** *instructor*
group by *dept_name*) **where** *avg_salary* > 42000;

We can give the subquery result relation a name, and rename the attributes, using the **as** clause, as illustrated below.
SQL> select *dept_name*, *avg_salary* **from** (**select** *dept_name*, **avg** (*salary*) **from** *instructor*
group by *dept_name*) **as** *dept_avg* (*dept_name*, *avg_salary*) **where** *avg_salary* > 42000;

The subquery result relation is named *dept avg*, with the attributes *dept_name* and *avg_salary*.

As another example, suppose we wish to find the maximum across all departments of the total salary at each department.
SQL> select max (*tot_salary*) **from** (**select** *dept_name*, **sum**(*salary*) **from** *instructor* **group by** *dept_name*) **as** *dept_total* (*dept_name*, *tot_salary*);

A subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the **from** clause. For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:
SQL> select *name*, *salary*, *avg_salary* **from** *instructor I1*, **lateral** (**select avg**(*salary*) as *avg_salary* **from** *instructor I2* **where** *I2.dept_name= I1.dept_name*);

The with Clause:

The **with** clause provides away of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

SQL> with *max_budget* (*value*) **as** (**select max**(*budget*) **from** *department*) **select** *budget* **from** *department*, *max budget* **where** *department.budget = max_budget.value*;

Scalar Subqueries:

wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**.
For Example: " lists all departments along with the number of instructors in each department:

SQL> select *dept_name*, (**select count**(*) **from** *instructor* **where** *department.dept_name = instructor.dept_name*) **as** *num_instructors* **from** *department*;

Modification of the Database

Now, we show how to add, remove, or change information with SQL.

Deletion:

A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

delete from r where P;

Note that a delete command operates on only one relation. If we want to delete tuples from several relations, we must use one delete command for each relation. The predicate in the where clause may be as complex as a select command's where clause. At the other extreme, the where clause may be empty.

Q: Delete all tuples in the instructor relation pertaining to instructors in the Finance department.

SQL> delete from *instructor where* dept name= *'Finance'*;

Q: Delete all instructors with a salary between $13,000 and $15,000.

SQL> delete from *instructor* **where** *salary between 13000 and 15000;*

although we may delete tuples from only one relation at a time, we may reference any number of relations in a select-from-where nested in the where clause of a delete. The delete request can contain a nested select that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all instructors with salary below the average at the university. We could write:

SQL> delete from *instructor* **where** *salary*< (*select* **avg** (*salary*) **from** *instructor*);

Insertion:

Insertion To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes. The simplest insert statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title "Database Systems", and 4 credit hours. We write:

SQL> insert into *course* **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

SQL allows the attributes to be specified as part of the insert statement. For example, the following SQL insert statements are identical in function to the preceding one:

SQL> insert into *course* (*course_id, title, dept name, credits*) **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

SQL> insert into *course* (*title, course_ id, credits, dept name*) **values** ('Database Systems', 'CS-437', 4, 'Comp. Sci.');

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of $18,000. We write:

SQL> insert into *instructor select ID, name, dept name,* 18000 **from** *student* **where** *dept name* = 'Music' and *tot cred* > 144;

the insert statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by null. Consider the request:

SQL>insert into *student* **values** ('3003', 'Green', 'Finance', null);

The tuple inserted by this request specified that a student with ID "3003" is in the Finance department, but the tot cred value for this student is not known. Consider the query:

SQL>select *student* **from** *student* **where** *tot cred* > 45;

Since the tot cred value of student "3003" is not known, we cannot determine whether it is greater than 45.

Updates:

In certain situations, we may wish to change a value in a tuple without changing all values in the tuple. For this purpose, the update statement can be used. As we could for insert and delete, we can choose the tuples to be updated by using a query. Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent. We write:

update instructor set salary= salary * 1.05;

The preceding update statement is applied once to each of the tuples in instructor relation. If a salary increase is to be paid only to instructors with salary of less than $70,000, we can write:

update instructor set salary = salary * 1.05 where salary < 70000;

Q:Give a 5 percent salary raise to instructors whose salary is less than average.

SQL>update *instructor* **set** *salary* = *salary* * 1.05 **where** *salary* < (**select avg** (*salary*) **from** *instructor*);

SQL provides a case construct that we can use to perform both the updates with a single update statement, avoiding the problem with the order of updates

SQL>update *instructor*
set *salary* = **case**
when *salary* <= 100000 **then** *salary* * 1.05
else *salary* * 1.03
end

The general form of the case statement is as follows. case when pred1 then result1 when pred2 then result2 ... when predn then resultn else result 0 end

The general form of the case statement is as follows:

case
when pred1 **then** result1
when pred2 **then** result2 ...
when predn **then** resultn

else result0
end

Scalar subqueries are also useful in SQL update statements, where they can be used in the set clause. Consider an update where we set the tot cred attribute of each student tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is not 'F' or null. To specify this update, we need to use a subquery in the set clause, as shown below:

SQL>update *student S set tot cred* = (**select sum**(*credits*) **from takes natural join** *course* **where** *S.ID= takes.ID* **and** *takes.grade* <> 'F' **and** *takes.grade* is **not null**);

we could use another update statement to replace null values by 0; a better alternative is to replace the clause "select sum(credits)" in the preceding subquery by the following select clause using a case expression:

SQL> select case
when sum(*credits*) is **not null then sum**(*credits*)
else 0
end

Joint Expressions

The natural join: operation operates on two relations and produce a relation as the result.
" for all instructors in the university who have taught some course, find their names and the course ID of all courses they taught"

SQL> select name, course_id from instructor, teaches where instructor.ID=teaches.ID;

SQL> select name, course_id from instructor natural join teaches;

Both of the above queries generate the same result.

A from clause in SQL query can have multiple relations combined using natural join as shown here:

SQL> SELECT a1,a2,…an FROM r1 NATURAL JOIN r2 NATURAL JOIN….. NATURAL JOIN rm WHERE P;

Q: " list the names of instructors along with the titles of course that they teach."

SQL>SELECT name, title FROM instructor NATURAL JOIN teaches, course WHERE teaches.course_id = course.course_id;

The operation JOIN .. USING:
Consider the operation: r1 JOIN r2 USING (a1, a2);

SQL>SELECT name, title FROM (instructor NATURAL JOIN teaches) JOIN course USING (course_id);

SQL>SELECT * FROM instructor JOIN teaches;

SQL>SELECT * FROM instructor JOIN teaches USING (course_id);

We can retrieve data from more than one tables using the JOIN statement. There are mainly 4 different types of JOINS in SQL server.

INNER JOIN/simple join
LEFT OUTER JOIN/LEFT JOIN
RIGHT OUTER JOIN/RIGHT JOIN
FULL OUTER JOIN

INNER JOIN

This type of SQL server JOIN returns rows from all tables in which the join condition is true. It takes the following syntax:

SELECT columns FROM table_1 INNER JOIN table_2 ON table_1.column = table_2.column;
We will use the following two tables to demonstrate this:

Students Table:

	admission	firstName	lastName	age
1	3420	Nicholas	Samuel	14
2	3380	Joel	John	15
3	3410	Japheth	Becky	16
4	3398	George	Joshua	14
5	3386	John	Lucky	15
6	3403	Simon	Dan	13
7	3400	Calton	Becham	16

Fee table:

	admission	course	amount_paid
1	3380	Electrical	20000
2	3420	ICT	15000
3	3398	Commerce	13000
4	3410	HR	12000

The following command demonstrates an INNER JOIN in SQL server with example:

SELECT Students.admission, Students.firstName, Students.lastName, Fee.amount_paid
FROM Students
INNER JOIN Fee
ON Students.admission = Fee.admission

The command returns the following:

	admission	firstName	lastName	amount_paid
1	3420	Nicholas	Samuel	15000
2	3380	Joel	John	20000
3	3410	Japheth	Becky	12000
4	3398	George	Joshua	13000

We can tell the students who have paid their fee. We used the column with common values in both tables, which is the admission column.

LEFT OUTER JOIN

This type of join will return all rows from the left-hand table plus records in the right-hand table with matching values. For example:

```
SELECT Students.admission, Students.firstName, Students.lastName, Fee.amount_paid
FROM Students
LEFT OUTER JOIN Fee
ON Students.admission = Fee.admission
```

The code returns the following:

	admission	firstName	lastName	amount_paid
1	3420	Nicholas	Samuel	15000
2	3380	Joel	John	20000
3	3410	Japheth	Becky	12000
4	3398	George	Joshua	13000
5	3386	John	Lucky	NULL
6	3403	Simon	Dan	NULL
7	3400	Calton	Becham	NULL

The records without matching values are replaced with NULLs in the respective columns.

RIGHT OUTER JOIN

This type of join returns all rows from the right-hand table and only those with matching values in the left-hand table. For example:

```
SELECT Students.admission, Students.firstName, Students.lastName, Fee.amount_paid
FROM Students
RIGHT OUTER JOIN Fee
ON Students.admission = Fee.admission
```

The statement for OUTER JOINS SQL server returns the following:

	admission	firstName	lastName	amount_paid
1	3380	Joel	John	20000
2	3420	Nicholas	Samuel	15000
3	3398	George	Joshua	13000
4	3410	Japheth	Becky	12000

The reason for the above output is that all rows in the Fee table are available in the Students table when matched on the admission column.

FULL OUTER JOIN

This type of join returns all rows from both tables with NULL values where the JOIN condition is not true. For example:

```
SELECT Students.admission, Students.firstName, Students.lastName, Fee.amount_paid
FROM Students
FULL OUTER JOIN Fee
ON Students.admission = Fee.admission
```

The code returns the following result for FULL OUTER JOINS queries in SQL:

	admission	firstName	lastName	amount_paid
1	3420	Nicholas	Samuel	15000
2	3380	Joel	John	20000
3	3410	Japheth	Becky	12000
4	3398	George	Joshua	13000
5	3386	John	Lucky	NULL
6	3403	Simon	Dan	NULL
7	3400	Calton	Becham	NULL

Join Types and Conditions:

To distinguish normal joins from outer joins, normal joins are called inner joins in SQL. The default join type, when the join clause is used without the outer prefix is the inner join. Thus,

SQL>select * from student join takes using (ID);

is equivalent to:

SQL>select * from student inner join takes using (ID);

Similarly, natural join is equivalent to natural inner join.

Views

SQL allows a "virtual relation" to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.

Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**.

View Definition:

We define a view in SQL by using the create view command. To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is:

create view v as <query expression>;

where is any legal query expression. The view name is represented by v.

EXAMPLE:

The clerk should not be authorized to access the instructor relation. Instead, a view relation faculty can be made available to the clerk, with the view defined as follows:

SQL>create view faculty as select ID, name, dept_name from instructor;

To create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section, we write:

```
SQL>create view physics_fall_2009 as
        select course.course_id, sec_id, building, room_number
        from course, section
        where course.course_id = section.course_id
                and course.dept_name = 'Physics'
                and section.semester = 'Fall'
                and section.year = '2009';
```

Using Views in SQL Queries:

Using the view physics fall 2009, we can find all Physics courses offered in the Fall 2009 semester in the Watson building by writing:

```
SQL>select course id from physics_fall_2009 where building= 'Watson';
```

View names may appear in a query any place where a relation name may appear, The attribute names of a view can be specified explicitly as follows:

```
SQL>create view departments_total_salary(dept_name, total_salary) as
        select dept_name, sum (salary)
        from instructor
        group by dept_name;
```

One view may be used in the expression defining another view. For example, we can define a view physics fall 2009 watson that lists the course ID and room number of all Physics courses offered in the Fall 2009 semester in the Watson building as follows:

```
SQL>create view physics_fall_2009 watson as
        select course_id, room_number
        from physics_fall_2009
        where building= 'Watson';
```

Materialized Views:

If the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

The process of keeping the materialized view up-to-date is called materialized view maintenance (or often, just view maintenance).

Update of a View:

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database. Suppose the view faculty, which we saw earlier, is made available to a clerk. Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

```
SQL>insert into faculty values ('30765', 'Green', 'Music');
```

However, to insert a tuple into instructor, we must have some value for salary. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple ('30765', 'Green', 'Music', null) into the instructor relation.

In general, an SQL view is said to be updatable (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The from clause has only one database relation.

• The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
• Any attribute not listed in the select clause can be set to null; that is, it does not have a not null constraint and is not part of a primary key.
• The query does not have a group by or having clause.

Under these constraints, the update, insert, and delete operations would be allowed on the following view:

```
SQL>create view history_instructors as
        select * from instructor where dept_name= 'History';
```

Transactions

A transaction consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

Once a transaction has executed commit work, its effects can no longer be undone by rollback work.

For instance, consider a banking application, where we need to transfer money from one bank account to another in the same bank. To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other. If the system crashes after subtracting the amount from the first account, but before adding it to the second account, the bank balances would be inconsistent. A similar problem would occur, if the second account is credited before subtracting the amount from the first account, and the system crashes just after crediting the amount.

Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database. Examples of integrity constraints are:
• An instructor name cannot be null.
• No two instructors can have the same instructor ID.
• Every department name in the course relation must have a matching department name in the department relation.
• The budget of a department must be greater than $0.00.

Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations. However, integrity constraints

can also be added to an existing relation by using the command **alter table table-name add constraint**, where constraint can be any constraint on the relation. When such a command is executed, the system first ensures that the relation satisfies the specified constraint. If it does, the constraint is added to the relation; if not, the command is rejected.

Constraints on a Single Relation:

The create table command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the create table command. The allowed integrity constraints include

- **not null**
- **unique**
- **check(<predicate>)**

We cover each of these types of constraints in the following sections.

Not Null Constraint:

Consider a tuple in the student relation where name is null. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be null. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes name and budget to exclude null values, by declaring it as follows:

name varchar(20) not null
budget numeric(12,2) not null

Unique Constraint:

SQL also supports an integrity constraint:

unique (Aj1 , Aj2 ,..., Ajm)

The unique specification says that attributes Aj1 , Aj2 ,..., Ajm form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be not null. Recall that a null value does not equal any other value.

The check Clause:

A common use of the check clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause check (budget > 0) in the create table command for relation department would ensure that the value of budget is nonnegative. As another example, consider the following:

SQL>create table_section (course_id varchar (8), sec_id varchar (8), semester varchar (6), year numeric (4,0), building varchar (15), room_number varchar (7), time_slot_id varchar (4), primary key (course_id, sec_id, semester, year), check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));

Here, we use the check clause to simulate an enumerated type, by specifying that semester must be one of 'Fall', 'Winter', 'Spring', or 'Summer'.

Referential Integrity:

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity.

More generally, let r1 and r2 be relations whose set of attributes are R1 and R2, respectively, with primary keys K1 and K2. We say that a subset of R2 is a foreign key referencing K1 in relation r1 if it is required that, for every tuple t2 in r2, there must be a tuple t1 in r1 such that t1.K1 = t2.α. Requirements of this form are called referential-integrity constraints, or subset dependencies.28
We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

dept_name varchar(20) references department

EXAMPLES:

```
SQL>create table department
        (dept_name varchar (20),
        building varchar (15),
        budget numeric (12,2) check (budget > 0),
        primary key (dept name));
SQL>create table course
        (course_id varchar (8),
        title varchar (50),
        dept_name varchar (20),
        credits numeric (2,0) check (credits > 0),
        primary key (course_id),
        foreign key (dept_name) references department);
```

Consider this definition of an integrity constraint on the relation course:

```
create table course
        ( ...
        foreign key (dept_name) references department
                on delete cascade
                on update cascade,
        ... );
```

Because of the clause on delete cascade associated with the foreign-key declaration, if a delete of a tuple in department results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the course relation, deleting the tuple that refers to the department that was deleted.

Integrity Constraint Violation During a Transaction:

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation.

To handle such situations, the SQL standard allows a clause initially deferred to be added to a constraint specification; the constraint would then be checked at the end of a transaction, and not at intermediate steps. A constraint can alternatively be specified as deferrable, which means it is checked immediately by default, but can be deferred when desired. For constraints declared as deferrable, executing a statement set constraints constraint-list deferred as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction.

Complex Check Conditions and Assertions:

As defined by the SQL standard, the predicate in the check clause can be an arbitrary predicate, which can include a subquery.

check (time_slot_id in (select time_slot_id from time_slot))

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions.

However, there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

• For each tuple in the student relation, the value of the attribute tot cred must equal the sum of credits of courses that the student has completed successfully.

• An instructor cannot teach in two different classrooms in a semester in the same time slot.

An assertion in SQL takes the form:

create assertion <assertion-name> check <predicate>;

An assertion example:

```
SQL>create assertion credits_earned_constraint check
        (not exists (select ID
                     from student
                     where tot_cred <> (select sum(credits)
                     from takes natural join course
                     where student.ID= takes.ID
                          and grade is not null and grade<> 'F' )
```

SQL Data Types and Schemas

we covered a number of built-in data types supported in SQL, such as integer types, real types, and character types.

Date and Time Types in SQL:

The SQL standard supports several data types relating to dates and times:

• **date:** A calendar date containing a (four-digit) year, month, and day of the month.

• **time:** The time of day, in hours, minutes, and seconds. A variant, time(p), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying time with timezone.

• **timestamp:** A combination of date and time. A variant, timestamp(p), can be used to specify the number of fractional digits for seconds (the default here being 6). Time-zone information is also stored if with timezone is specified.

Date and time values can be specified like this:

```
date '2001-04-25'
time '09:30:00'
timestamp '2001-04-25 10:29:01.45'
```

Default Values:

SQL allows a default value to be specified for an attribute as illustrated by the following create table statement:

```
SQL>create table student
(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
```

tot_cred numeric (3,0) default 0, primary key (ID));

The default value of the tot cred attribute is declared to be 0. As a result, when a tuple is inserted into the student relation, if no value is provided for the tot cred attribute, its value is set to 0. The following insert statement illustrates how an insertion can omit the value for the tot cred attribute.

SQL>insert into student(ID, name, dept_name) values ('12789', 'Newman', 'Comp. Sci.');

Index Creation:

An index on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.

For example, if we create in index on attribute ID of relation student, the database system can find the record with any specified ID value, such as 22201, or 44553, directly, without reading all the tuples of the student relation. An index can also be created on a list of attributes, for example on attributes name, and dept_name of student.

Syntax:

create index studentID_index on student(ID);

The above statement creates an index named studentID_index on the attribute ID of the relation student.

Large-Object Types:

Many current-generation database applications need to store attributes that can be large (of the order of many kilobytes), such as a photograph, or very large (of the order of many megabytes or even gigabytes), such as a high-resolution medical image or video clip.

SQL therefore provides large-object data types for character data (clob) and binary data (blob). The letters "lob" in these data types stand for "Large OBject."

For example, we may declare attributes

book review clob(10KB)
image blob(10MB)
movie blob(2GB)

It is inefficient or impractical to retrieve an entire large object into memory. Instead, an application would usually use an SQL query to retrieve a "locator" for a large object and then use the locator to manipulate the object from the host language in which the application itself is written. For instance, the **JDBC** application program interface permits a locator to be fetched instead of the entire large object; the locator can then be used to fetch the large object in small pieces, rather than all at once, much like reading data from an operating system file using a read function call.

User-Defined Types:

SQL supports two forms of user-defined data types.

i) distinct types.

ii) structured data types, allows the creation of complex data types with nested record structures, arrays, and multisets.

The **create type** clause can be used to define new types. For example, the statements:

create type Dollars as numeric(12,2) final;
create type Pounds as numeric(12,2) final;

define the user-defined types Dollars and Pounds to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point. The newly created types can then be used, for example, as types of attributes of relations. For example, we can declare the department table as:

SQL>create table department (dept name varchar (20),
building varchar (15),
budget Dollars);

As a result of strong type checking, the expression (department.budget+20) would not be accepted since the attribute and the integer constant 20 have different types. Values of one type can be cast (that is, converted) to another domain, as illustrated below: ‘

cast (department.budget to numeric(12,2))

SQL provides **drop type** and **alter type** clauses to drop or modify types that have been created earlier.

SQL had a similar but subtly different notion of **domain** (introduced in SQL-92), which can add integrity constraints to an underlying type. For example, we could define a domain DDollars as follows.

create domain DDollars as numeric(12,2) not null;

The domain DDollars can be used as an attribute type, just as we used the type Dollars. However, there are two significant differences between types and domains:

1. Domains can have constraints, such as not null, specified on them, and can have default values defined for variables of the domain type, whereas userdefined types cannot have constraints or default values specified on them. User-defined types are designed to be used not just for specifying attribute types, but also in procedural extensions to SQL where it may not be possible to enforce constraints.
2. Domains are not strongly typed. As a result, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

When applied to a domain, the check clause permits the schema designer to specify a predicate that must be satisfied by any attribute declared to be from this domain. For instance, a check clause can ensure that an instructor’s salary domain allows only values greater than a specified value:

SQL>create domain YearlySalary numeric(8,2) constraint salary_value_test check(value >= 29000.00);

As another example, a domain can be restricted to contain only a specified set of values by using the in clause:

SQL>create domain degree_level varchar(10) constraint degree_level_test check (value in (’Bachelors’, ’Masters’, or ’Doctorate’));

Create Table Extensions:

Applications often require creation of tables that have the same schema as an existing table. SQL provides a create table like extension to support this task:

SQL>create table temp_instructor like instructor;

The above statement creates a new table temp instructor that has the same schema as instructor.
When writing a complex query, it is often useful to store the result of a query as a new table; the table is usually temporary.
For example the following statement creates a table t1 containing the results of a query.

SQL>create table t1 as (select * from instructor where dept_name= ’Music’) with data;

Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a privilege.

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected.

Granting and Revoking of Privileges:

The SQL standard includes the privileges select, insert, update, and delete. The privilege all privileges can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically.

The SQL data-definition language includes commands to grant and revoke privileges. The grant statement is used to confer authorization. The basic form of this statement is:

grant <privilege list>
on< relation name or view name>
To< user/role list>

The **select authorization** on a relation is required to read tuples in the relation. The following grant statement grants database users Amit and Satoshi select authorization on the department relation:

grant select on department to Amit, Satoshi;

This allows those users to run queries on the department relation.

The **update authorization** on a relation allows a user to update any tuple in the relation. The update authorization may be given either on all attributes of the relation or on only some.

This grant statement gives users Amit and Satoshi update authorization on the budget attribute of the department relation:

grant update (budget) on department to Amit, Satoshi;

The **insert authorization** on a relation allows a user to insert tuples into the relation.

The **delete authorization** on a relation allows a user to delete tuples from a relation.

The user name public refers to all current and future users of the system. Thus, privileges granted to public are implicitly granted to all current and future users.

To **revoke** an authorization, we use the revoke statement. It takes a form almost identical to that of grant:

revoke < privilege list>
on< relation name or view name>
from< user/role list>;

Thus, to revoke the privileges that we granted previously, we write

revoke select on department from Amit, Satoshi;
revoke update (budget) on department from Amit, Satoshi;

Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user.

Roles:

Consider the real-world roles of various people in a university. Each instructor must have the same types of authorizations on the same set of relations. Whenever a new instructor is appointed, she will have to be given all these authorizations individually.

The notion of roles captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users.

In our university database, examples of roles could include instructor, teaching assistant, student, dean, and department chair.

Roles can be created in SQL as follows:

create role instructor;

Roles can then be granted privileges just as the users can, as illustrated in this statement:

grant select on takes to instructor;

Roles can be granted to users, as well as to other roles, as these statements show:

grant dean to Amit;
create role dean;
grant instructor to dean;
grant dean to Satoshi;

Thus the privileges of a user or a role consist of:

- All privileges directly granted to the user/role.
- All privileges granted to roles that have been granted to the user/role.

Authorization on Views:

This view can be defined in SQL as follows:

SQL>create view geo_instructor as (select * from instructor where dept_name = 'Geology');

Suppose that the staff member issues the following SQL query:

SQL>select * from geo_instructor;

Clearly, the staff member is authorized to see the result of this query. A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user who creates a view cannot be given update authorization on a view without having update authorization on the relations used to define the view.

Authorizations on Schema:

The SQL standard specifies a primitive authorization mechanism for the database schema: Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices.

However, SQL includes a references privilege that permits a user to declare foreign keys when creating relations. The SQL references privilege is granted on specific attributes in a manner like that for the update privilege. The following grant statement allows user Mariano to create relations that reference the key branch name of the branch relation as a foreign key: '

SQL>grant references (dept_name) on department to Mariano;

Transfer of Privileges:

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users.

For example, if we wish to allow Amit the select privilege on department and allow Amit to grant this privilege to others, we write:

SQL>grant select on department to Amit with grant option;

Assume that, initially, the database administrator grants update authorization on teaches to users U1, U2, and U3, who may in turn pass on this authorization to other users. The passing of a specific authorization from one user to another can be represented by an authorization graph. The nodes of this graph are the users.

Consider the graph for update authorization on teaches. The graph includes an edge Ui → Uj if user Ui grants update authorization on teaches to Uj . The root of the graph is the database administrator.

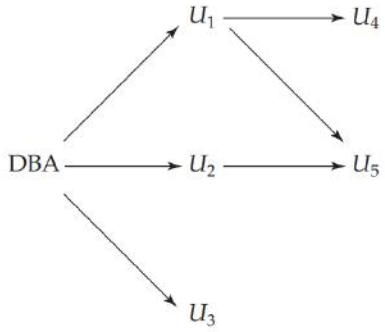


Figure 4.10 Authorization-grant graph ($U_1, U_2, \ldots, U_5$ are users and DBA refers to the database administrator).

Observe that user U5 is granted authorization by both U1 and U2; U4 is granted authorization by only U1.

Revoking of Privileges:

Revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called cascading revocation. In most database systems, cascading is the default behavior. However, the revoke statement may specify restrict in order to prevent cascading revocation:

SQL>revoke select on department from Amit, Satoshi restrict;

The following revoke statement revokes only the grant option, rather than the actual select privilege:

SQL>revoke grant option for select on department from Amit;

Note that some database implementations do not support the above syntax; instead, the privilege itself can be revoked, and then granted again without the grant option.

To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

granted by current_role

to the grant statement, provided the current role is not null.

Advanced SQL

Accessing SQL From a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

- **Dynamic SQL:** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The dynamic SQL component of SQL allows programs to construct and submit SQL queries at runtime.

- **Embedded SQL:** Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor. The preprocessor submits the SQL statements to the database system for precompilation and optimization; then it replaces the SQL statements in the application program with appropriate code and function calls before invoking the programming-language compiler.

JDBC (Java Database Connectivity):

The JDBC standard defines an application program interface (API) that Java programs can use to connect to database servers.

```
public static void JDBCexample(String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                    rset.getFloat(2));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

An Example of JDBC code

The Java program must import java.sql.*, which contains the interface definitions for the functionality provided by JDBC.

Connecting to the Database:

A connection is opened using the getConnection method of the DriverManager class (within java.sql). This method takes three parameters.

• The first parameter to the getConnection call is a string that specifies the URL, or machine name, where the server runs (in our example, db.yale.edu), along with possibly some other information such as the protocol to be used to communicate with the database (in our example, jdbc:oracle:thin:; we shall shortly see why this is required), the port number the database system uses for communication (in our example, 2000), and the specific database on the server to be used (in our example, univdb). Note that JDBC specifies only the API, not the communication protocol. A JDBC driver may support multiple protocols, and we must specify one supported by both the database and the driver. The protocol details are vendor specific.

• The second parameter to getConnection is a database user identifier, which is a string.

• The third parameter is a password, which is also a string. (Note that the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.)

In our example in the figure, we have created a Connection object whose handle is conn.

Shipping SQL Statements to the Database System:

Once a database connection is open, the program can use it to send SQL statements to the database system for execution. This is done via an instance of the class Statement. A Statement object is not the SQL statement itself, but rather an object that allows the Java program to invoke methods that ship an SQL statement given as an argument for execution by the database system.

Our example creates a Statement handle (stmt) on the connection conn. To execute a statement, we invoke either the executeQuery method or the executeUpdate method, depending on whether the SQL statement is a query (and, thus, returns a result set) or nonquery statement such as update, insert, delete, create table, etc. In our example, stmt.executeUpdate executes an update statement that inserts into the instructor relation. It returns an integer giving the number of tuples inserted, updated, or deleted. For DDL statements, the return value is zero. The try { ... } catch { ... } construct permits us to catch any exceptions (error conditions) that arise when JDBC calls are made, and print an appropriate message to the user.

Retrieving the Result of a Query:

The example program executes a query by using stmt.executeQuery. It retrieves the set of tuples in the result into a ResultSet object rset and fetches them one tuple at a time. The next method on the result set tests whether or not there remains at least one unfetched tuple in the result set and if so, fetches it. The return value of the next method is a Boolean indicating whether it fetched a tuple. Attributes from the fetched tuple are retrieved using various methods whose names begin with get. The method getString can retrieve any of the basic SQL data types (converting the value to a Java String object), but more restrictive methods such as getFloat can be used as well. The argument to the various get methods can either be an attribute name specified as a string, or an integer indicating the position of the desired attribute within the tuple. Figure 5.1 shows two ways of retrieving the values of attributes in a tuple: using the name of the attribute (dept name) and using the position of the attribute (2, to denote the second attribute).

Prepared Statements:

We can create a prepared statement in which some values are replaced by “?”, thereby specifying that actual values will be provided later. The database system compiles the query when it is prepared. Each time the query is executed (with new values to replace the “?”s), the database system can reuse the previously compiled form of the query and apply the new values.

```
PreparedStatement pStmt = conn.prepareStatement(
                    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

Figure 5.2 Prepared statements in JDBC code.

The prepareStatement method of the Connection class submits an SQL statement for compilation. It returns an object of class PreparedStatement. At this point, no SQL statement has been executed. The executeQuery and executeUpdate methods of PreparedStatement class do that. But before they can be invoked, we must use methods of class PreparedStatement that assign values for the "?" parameters. The setString method and other similar methods such as setInt for other basic SQL types allow us to specify the values for the parameters.

In our example, suppose that the values for the variables ID, name, dept name, and salary have been entered by a user, and a corresponding row is to be inserted into the instructor relation. Suppose that, instead of using a prepared statement, a query is constructed by concatenating the strings using the following Java expression:

"insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ")"

and the query is executed directly using the executeQuery method of a Statement object.

Suppose a Java program inputs a string name and constructs the query:

"select * from instructor where name = '" + name + "'"

If the user, instead of entering a name, enters:

X' or 'Y' = 'Y

then the resulting statement becomes:

"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"

which is:

select * from instructor where name = 'X' or 'Y' = 'Y'

The input string would have escape characters inserted, so the resulting query becomes:

"select * from instructor where name = 'X\' or \'Y\' = \'Y'

which is harmless and returns the empty relation.

Callable Statements:

JDBC also provides a CallableStatement interface that allows invocation of SQL stored procedures and functions (described later, in Section 5.2). These play the same role for functions and procedures as prepareStatement does for queries.

CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");

CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");

The data types of function return values and out parameters of procedures must be registered using the method registerOutParameter(), and can be retrieved using get methods similar to those for result sets. See a JDBC manual for more details.

Metadata Features:

The Java code segment below uses JDBC to print out the names and types of all columns of a result set. The variable rs in the code below is assumed to refer to a ResultSet instance obtained by executing a query.

```
ResultSetMetaData rsmd = rs.getMetaData();

for(int i = 1; i <= rsmd.getColumnCount(); i++) {

        System.out.println(rsmd.getColumnName(i));

        System.out.println(rsmd.getColumnTypeName(i));

}
```

The getColumnCount method returns the arity (number of attributes) of the result relation.

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
      // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
      //          and Column-Pattern
      // Returns: One row for each column; row has a number of attributes
      //          such as COLUMN_NAME, TYPE_NAME
while( rs.next()) {
      System.out.println(rs.getString("COLUMN_NAME"),
                rs.getString("TYPE_NAME");
}
```

Figure 5.3 Finding column information in JDBC using `DatabaseMetaData`.

Other Features:

JDBC provides a number of other features, such as updatable result sets. It can create an updatable result set from a query that performs a selection and/or a projection on a database relation.

The method setAutoCommit() in the JDBC Connection interface allows this behavior to be turned on or off. Thus, if conn is an open connection, conn.setAutoCommit(false) turns off automatic commit. Transactions must then be committed or rolled back explicitly using either conn.commit() or conn.rollback(). conn.setAutoCommit(true) turns on automatic commit.

To fetch large objects, the ResultSet interface provides methods getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively.

Conversely, to store large objects in the database, the PreparedStatement class permits a database column whose type is blob to be linked to an input stream (such as a file that has been opened) using the method setBlob(int parameterIndex, InputStream inputStream).

JDBC includes a row set feature that allows result sets to be collected and shipped to other applications. Row sets can be scanned both backward and forward and can be modified.

ODBC:

The Open Database Connectivity (ODBC) standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC.

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    {
        char deptname[80];
        float salary;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * sqlquery = "select dept_name, sum (salary)
                           from instructor
                           group by dept_name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf (" %s %g\n", depthname, salary);
            }
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Figure 5.4 ODBC code example.

Figure 5.4 shows an example of C code using the ODBC API. The first step in using ODBC to communicate with a server is to set up a connection with the server. To do so, the program first allocates an SQL environment, then a database connection handle. ODBC defines the types HENV, HDBC, and RETCODE. The program then opens the database connection by using SQLConnect.

Once the connection is set up, the program can send SQL commands to the database by using SQLExecDirect.

The SQLFetch statement is in a while loop that is executed until SQLFetch returns a value other than SQL SUCCESS.

The ODBC standard defines conformance levels, which specify subsets of the functionality defined by the standard. An ODBC implementation may provide only core level features, or it may provide more advanced (level 1 or level 2) features. Level 1 requires support for fetching information about the catalog, such as information about what relations are present and the types of their attributes. Level 2 requires further features, such as the ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

The SQL standard defines a call level interface (CLI) that is similar to the ODBC interface.

Embedded SQL:

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, C++, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language constitute embedded SQL.

To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form:

EXEC SQL< embedded SQL statement>;

Before executing any SQL statements, the program must first connect to the database. This is done using:

EXEC SQL connect to server user user-name using password;

Here, server identifies the server to which a connection is to be established.

The syntax for declaring the variables, however, follows the usual host language syntax.

```
EXEC SQL BEGIN DECLARE SECTION;
        int credit amount;
EXEC SQL END DECLARE SECTION;
```

we wish to find the names of all students who have taken more than credit amount credit hours. We can write this query as follows:

```
EXEC SQL
        declare c cursor for
        select ID, name
        from student
        where tot_cred > :credit amount;
```

The variable c in the preceding expression is called a cursor for the query. We use this variable to identify the query. We then use the open statement, which causes the query to be evaluated.

The open statement for our sample query is as follows:

EXEC SQL open c;

For our example query, we need one variable to hold the ID value and another to hold the name value. Suppose that those variables are si and sn, respectively, and have been declared within a DECLARE section. Then the statement:

EXEC SQL fetch c into :si, :sn;

We must use the close statement to tell the database system to delete the temporary relation that held the result of the query. For our example, this statement takes the form

EXEC SQL close c;

A database-modification request takes the form

EXEC SQL < any valid update, insert, or delete>;

Database relations can also be updated through cursors. For example, if we want to add 100 to the salary attribute of every instructor in the Music department, we could declare a cursor as follows.

```
EXEC SQL
        declare c cursor for
        select *
        from instructor
        where dept name= 'Music' for update;
```

after fetching each tuple we execute the following code:

```
EXEC SQL
        update instructor
        set salary = salary + 100
        where current of c;
```

Transactions can be committed using EXEC SQL COMMIT, or rolled back using EXEC SQL ROLLBACK.

Functions and Procedures

Procedures and functions allow "business logic" to be stored in the database, and executed from SQL statements. While such business logic can be encoded as programming-language procedures stored entirely outside the database, defining them as stored procedures in the database has several advantages.

For example, the procedural languages supported by Oracle (PL/SQL), Microsoft SQL Server (TransactSQL), and PostgreSQL (PL/pgSQL) all differ from the standard syntax we present here.

```
create function dept_count(dept_name varchar(20))
    returns integer
    begin
    declare d_count integer;
        select count(*) into d_count
        from instructor
        where instructor.dept_name= dept_name
    return d_count;
    end
```

Figure 5.5 Function defined in SQL.

Declaring and Invoking SQL Functions and Procedures:

This function can be used in a query that returns names and budgets of all departments with more than 12 instructors:

SQL>select dept_name, budget from instructor where dept_count(dept_name) > 12;

The SQL standard supports functions that can return tables as results; such functions are called table functions.

The function can be used in a query as follows:

SQL>select * from table(instructor_of('Finance'));

```
create function instructors_of (dept_name varchar(20))
    returns table (
        ID varchar (5),
        name varchar (20),
        dept_name varchar (20),
        salary numeric (8,2))
return table
    (select ID, name, dept_name, salary
    from instructor
    where instructor.dept_name = instructor_of.dept_name);
```

Figure 5.6 Table function in SQL.

SQL also supports procedures. The dept_count function could instead be written as a procedure:

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out d_count integer)
  begin
      select count(*) into d_count
      from instructor
      where instructor.dept_name= dept_count_proc.dept_name
  end
```

The keywords in and out indicate, respectively, parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results. Procedures can be invoked either from an SQL procedure or from embedded SQL by the call statement:

declare d_count integer;

call dept_count_proc('Physics', d count);

Language Constructs for Procedures and Functions:

SQL supports constructs that give it almost all the power of a general-purpose programming language. The part of the SQL standard that deals with these constructs is called the **Persistent Storage Module (PSM).**

SQL:1999 supports the while statements and the repeat statements by the following syntax:

```
while boolean expression do
    sequence of statements;
end while

repeat
    sequence of statements;
until boolean expression
end repeat
```

There is also a **for** loop that permits iteration over all results of a query:

```
declare n integer default 0;
for r as
        select budget from department
        where dept_name = 'Music'
do
    set n = n− r.budget
end for
```

The conditional statements supported by SQL include if-then-else statements by using this syntax:

```
if boolean expression
    then statement or compound statement
elseif boolean expression
    then statement or compound statement
else statement or compound statement
end if
```

The SQL procedural language also supports the signaling of exception conditions, and declaring of handlers that can handle the exception, as in this code:

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
sequence of statements
end
```

External Language Routines:

External procedures and functions can be specified in this way (note that the exact syntax depends on the specific database system you use):

```
create procedure dept_count_proc( in dept_name varchar(20),
                                  out count integer)
language C
external name '/usr/avi/bin/dept_count_proc'

create function dept_count (dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

If the code is written in a "safe" language such as Java or C#, there is another possibility: executing the code in a sandbox within the database query execution process itself. The sandbox allows the Java or C# code to access its own memory area, but prevents the code from reading or updating the memory of the query execution process, or accessing files in the file system.

Several database systems today support external language routines running in a sandbox within the query execution process. For example, Oracle and IBM DB2 allow Java functions to run as part of the database process. Microsoft SQL Server allows procedures compiled into the Common Language Runtime (CLR) to execute within the database process; such procedures could have been written, for example, in C# or Visual Basic. PostgreSQL allows functions defined in several languages, such as Perl, Python, and Tcl.

Triggers

A trigger is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
2. Specify the actions to be taken when the trigger executes.

Need for Triggers:

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL.

As an illustration, we could design a trigger that, whenever a tuple is inserted into the takes relation, updates the tuple in the student relation for the student taking the course by adding the number of credits for the course to the student's total credits.

Triggers in SQL:

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
        select time_slot_id
        from time_slot)) /* time_slot_id not present in time_slot */
begin
   rollback
end;

create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
        select time_slot_id
        from time_slot) /* last tuple for time_slot_id deleted from time_slot */
   and orow.time_slot_id in (
        select time_slot_id
        from section)) /* and time_slot_id still referenced from section*/
begin
   rollback
end;
```

Figure 5.8 Using triggers to maintain referential integrity.

For example, to specify that a trigger executes after an update to the grade attribute of the takes relation, we write:

after update of takes on grade

The referencing old row as clause can be used to create a variable storing the old value of an updated or deleted row. The referencing new row as clause can be used with updates in addition to inserts.

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
   update student
   set tot_cred= tot_cred+
          (select credits
           from course
           where course.course_id= nrow.course_id)
   where student.id = nrow.id;
end;
```

Figure 5.9 Using a trigger to maintain *credits_earned* values.

```
create trigger setnull before update on takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
   set nrow.grade = null;
end;
```

Figure 5.10 Example of using set to change an inserted value.

```
create trigger reorder after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
                    from minlevel
                    where minlevel.item = orow.item)
and orow.level > (select level
                  from minlevel
                  where minlevel.item = orow.item)
begin atomic
      insert into orders
            (select item, amount
             from reorder
             where reorder.item = orow.item);
end;
```

Figure 5.11 Example of trigger for reordering an item.

Triggers can be disabled or enabled; by default they are enabled when they are created, but can be disabled by using alter trigger trigger name disable (some databases use alternative syntax such as disable trigger trigger name). A trigger can instead be dropped, which removes it permanently, by using the command drop trigger trigger name.

When Not to Use Triggers:

For example, we could implement the on delete cascade feature of a foreign-key constraint by using a trigger, instead of using the cascade feature.

Modern database systems, however, provide built-in facilities for database replication, making triggers unnecessary for replication in most cases.

Another problem with triggers lies in unintended execution of the triggered action when data are loaded from a backup copy,6 or when database updates at a site are replicated on a backup site. In such cases, the triggered action has already been executed, and typically should not be executed again. When loading data, triggers can be disabled explicitly.

As an alternative, some database systems allow triggers to be specified as not for replication, which ensures that they are not executed on the backup site during database replication.

Triggers should be written with great care, since a trigger error detected at runtime causes the failure of the action statement that set off the trigger.

Triggers can serve a very useful purpose, but they are best avoided when alternatives exist. Many trigger applications can be substituted by appropriate use of stored procedures.

Recursive Queries

Suppose now that we want to find out which courses are a prerequisite whether directly or indirectly, for a specific course—say, CS-347. That is, we wish to find a course that is a direct prerequisite for CS-347, or is a prerequisite for a course that is a prerequisite for CS-347, and so on.

The transitive closure of the relation prereq is a relation that contains all pairs (cid, pre) such that pre is a direct or indirect prerequisite of cid. There are numerous applications that require computation of similar transitive closures on hierarchies.

Transitive Closure Using Iteration:

One way to write the above query is to use iteration: First find those courses that are a direct prerequisite of CS-347, then those courses that are a prerequisite of all the courses under the first set, and so on. This iterative process continues until we reach an iteration where no courses are added.

Figure 5.13 shows a function findAllPrereqs(cid) to carry out this task; the function takes the course id of the course as a parameter (cid), computes the set of all direct and indirect prerequisites of that course, and returns the set. The procedure uses three temporary tables:

- c_prereq: stores the set of tuples to be returned.
- new_c_prereq: stores the courses found in the previous iteration.
- temp: used as temporary storage while sets of courses are manipulated

Note that SQL allows the creation of temporary tables using the command create temporary table; such tables are available only within the transaction executing the query, and are dropped when the transaction finishes.

```
create function findAllPrereqs(cid varchar(8))
    -- Finds all courses that are prerequisite (directly or indirectly) for cid
returns table (course_id varchar(8))
    -- The relation prereq(course_id, prereq_id) specifies which course is
    -- directly a prerequisite for another course.
begin
    create temporary table c_prereq (course_id varchar(8));
        -- table c_prereq stores the set of courses to be returned
    create temporary table new_c_prereq (course_id varchar(8));
        -- table new_c_prereq contains courses found in the previous iteration
    create temporary table temp (course_id varchar(8));
        -- table temp is used to store intermediate results
    insert into new_c_prereq
        select prereq_id
        from prereq
        where course_id = cid;
    repeat
        insert into c_prereq
            select course_id
            from new_c_prereq;

        insert into temp
            (select prereq.course_id
                from new_c_prereq, prereq
                where new_c_prereq.course_id = prereq.prereq_id
            )
            except (
                select course_id
                from c_prereq
            );
        delete from new_c_prereq;
        insert into new_c_prereq
            select *
            from temp;
        delete from temp;

    until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end
```

Figure 5.13 Finding all prerequisites of a course.

The procedure inserts all direct prerequisites of course cid into new c_prereq before the repeat loop. The repeat loop first adds all courses in new c_prereq to c_prereq . Next, it computes prerequisites of all those courses in new c_prereq , except those that have already been found to be prerequisites of cid, and stores them in the temporary table temp. Finally, it replaces the contents of new c_prereq by the contents of temp. The repeat loop terminates when it finds no new (indirect) prerequisites.

Figure 5.14 shows the prerequisites that would be found in each iteration, if the procedure were called for the course named CS-347.

Iteration Number	Tuples in c1
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Figure 5.14 Prerequisites of CS-347 in iterations of function *findAllPrereqs*.

Recursion in SQL:

We can use recursion to define the set of courses that are prerequisites of a particular course, say CS-347, as follows. The courses that are prerequisites (directly or indirectly) of CS-347 are:

1. Courses that are prerequisites for CS-347.
2. Courses that are prerequisites for those courses that are prerequisites (directly or indirectly) for CS-347.

Any recursive view must be defined as the union of two subqueries: a base query that is nonrecursive and a recursive query that uses the recursive view. In the example in Figure 5.15, the base query is the select on prereq while the recursive query computes the join of prereq and rec prereq.

```
with recursive c_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select prereq.prereq_id, c_prereq.course_id
        from prereq, c_prereq
        where prereq.course_id = c_prereq.prereq_id
    )
select *
from c_prereq;
```

Figure 5.15 Recursive query in SQL.

In particular, recursive queries should not use any of the following constructs, since they would make the query nonmonotonic:

- Aggregation on the recursive view.
- not exists on a subquery that uses the recursive view.
- Set difference (except) whose right-hand side uses the recursive view.

SQL also allows creation of recursively defined permanent views by using create recursive view in place of with recursive.

OLAP**

An online analytical processing (OLAP) system is an interactive system that permits an analyst to view different summaries of multidimensional data. The word online indicates that an analyst must be able to request new summaries and get responses online, within a few seconds, and should not be forced to wait for a long time to see the result of a query.

There are many OLAP products available, including some that ship with database products such as Microsoft SQL Server, and Oracle, and other standalone tools.

Online Analytical Processing:

Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item name, color, and size, and that we have a relation sales with the schema.

sales (item_name, color, clothes_size, quantity)

Suppose that item name can take on the values (skirt, dress, shirt, pants), color can take on the values (dark, pastel, white), clothes size can take on values (small, medium, large), and quantity is an integer value representing the total number of items of a given {item name, color, clothes size }. An instance of the sales relation is shown in Figure 5.16.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2

Figure 5.16 An example of *sales* relation.

Given a relation used for data analysis, we can identify some of its attributes as measure attributes, since they measure some value, and can be aggregated upon. For instance, the attribute quantity of the sales relation is a measure attribute, since it measures the number of units sold. Some (or all) of the other attributes of the relation are identified as dimension attributes, since they define the dimensions on which measure attributes, and summaries of measure attributes, are viewed. In the sales relation, item name, color, and clothes size are dimension attributes.

Data that can be modeled as dimension attributes and measure attributes are called multidimensional data.

The table shows total quantities for different combinations of item name and color. The value of clothes size is specified to be all, indicating that the displayed values are a summary across all values of clothes size (that is, we want to group the “small”, “medium”, and “large” items into one single group. The table in Figure 5.17 is an example of a cross-tabulation (or cross-tab, for short), also referred to as a pivot-table. In general, a cross-tab is a table derived from a relation (say R), where values for one attribute of relation R (say A) form the row headers and values for another attribute of relation R (say B) form the column header. For example, in Figure 5.17, the attribute item name corresponds to A(with values “dark”, “pastel”, and “white”), and the attribute color corresponds to to B (with attributes “skirt”, “dress”, “shirt”, and “pants”). Each cell in the pivot-table can be identified by (ai, b j), where ai is a value for A and b j a value for B.

clothes_size **all**

item_name \ *color*	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

Figure 5.17 Cross tabulation of *sales* by *item_name* and *color*.

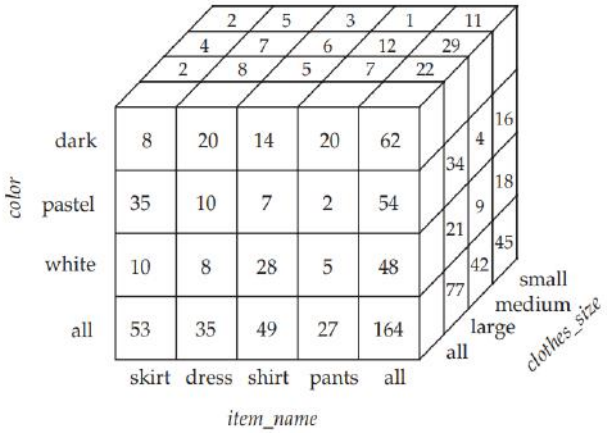


Figure 5.18 Three-dimensional data cube.

The generalization of a cross-tab, which is two-dimensional, to n dimensions can be visualized as an n-dimensional cube, called the data cube. Figure 5.18 shows a data cube on the sales relation. The data cube has three dimensions, item name, color, and clothes size, and the measure attribute is quantity. Each cell is identified by values for these three dimensions.

In the example of Figure 5.18, there are 3 colors, 4 items, and 3 sizes resulting in a cube size of 3 × 4 × 3 = 36. Including the summary values, we obtain a 4 × 5 × 4 cube, whose size is 80. In fact, for a table with n dimensions, aggregation can be performed with grouping on each of the 2n subsets of the n dimensions.

The operation of changing the dimensions used in a cross-tab is called pivoting. OLAP systems allow an analyst to see a cross-tab on item name and color for a fixed value of clothes size, for example, large, instead of the sum across all sizes. Such an operation is referred to as slicing, since it can be thought of as viewing a slice of the data cube. The operation is sometimes called dicing, particularly when values for multiple dimensions are fixed.

OLAP systems permit users to view data at any desired level of granularity. The operation of moving from finer-granularity data to a coarser granularity (by means of aggregation) is called a rollup. In our example, starting from the data cube on the sales table, we got our example cross-tab by rolling up on the attribute clothes size. The opposite operation—that of moving from coarser-granularity data to finer-granularity data—is called a drill down. Clearly, finer-granularity data cannot be generated from coarse-granularity data; they must be generated either from the original data, or from even finer-granularity summary data.

The different levels of detail for an attribute can be organized into a hierarchy. Figure 5.19a shows a hierarchy on the datetime attribute.

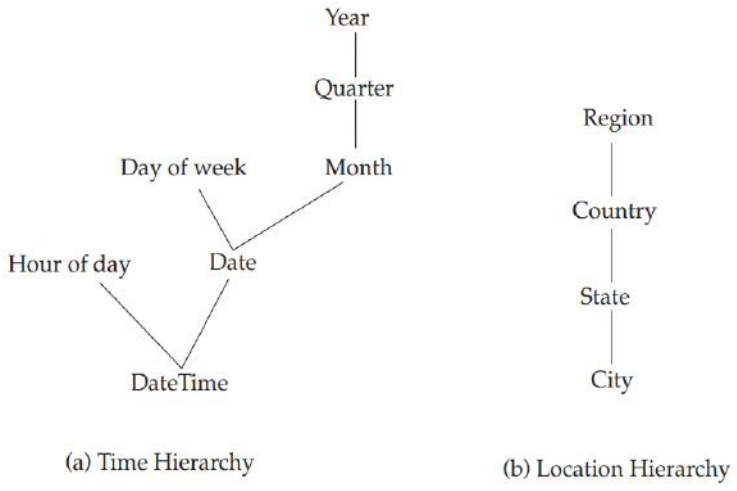


Figure 5.19 Hierarchies on dimensions.

Cross-Tab and Relational Tables:

A cross-tab with summary rows/columns can be represented by introducing a special value all to represent subtotals, as in Figure 5.21. The SQL standard actually uses the null value in place of all, but to avoid confusion with regular null values, we shall continue to use all.

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figure 5.20 Cross tabulation of *sales* with hierarchy on *item_name*.

The SQL standard actually uses the null value in place of all, but to avoid confusion with regular null values, we shall continue to use all.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	white	**all**	28
shirt	**all**	**all**	49
pants	dark	**all**	20
pants	pastel	**all**	2
pants	white	**all**	5
pants	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Figure 5.21 Relational representation of the data in Figure 5.17.

OLAP in SQL:

Several SQL implementations, such as Microsoft SQL Server, and Oracle, support a pivot clause in SQL, which allows creation of cross-tabs. Given the sales relation from Figure 5.16, the query:

```
select *
from sales
pivot (
    sum(quantity)
    for color in ('dark','pastel','white')
)
order by item_name;
```

item_name	clothes_size	dark	pastel	white
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2

Figure 5.22 Result of SQL pivot operation on the *sales* relation of Figure 5.16.

Consider again our retail shop example and the relation:

sales (item_name, color, clothes_size, quantity)

We can find the number of items sold in each item name by writing a simple group by query:

SQL>select item_name, sum(quantity) from sales group by item_name;

For example, we can find a breakdown of sales by item-name and color by writing:

SQL>select item_name, color, sum(quantity) from sales group by item_name, color;

item_name	color	quantity
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5

Figure 5.24 Query result.

{ (*item_name, color, clothes_size*), (*item_name, color*), (*item_name, clothes_size*), (*color, clothes_size*), (*item_name*), (*color*), (*clothes_size*), () }

The cube construct allows us to accomplish this in one query:

SQL>select item_name, color, clothes_size, sum(quantity)

from sales

group by cube(item name, color, clothes size);

The above query produces a relation whose schema is:

(item_name, color, clothes_size, sum(quantity))

It also uses all in place of null so as to be more readable to the average user. To generate that relation in SQL, we arrange to substitute all for null. The query:

SQL>select item_name, color, sum(quantity) from sales group by cube(item_name, color);

The rollup construct is the same as the cube construct except that rollup generates fewer group by queries.

SQL>select item_name, color, clothes_size, sum(quantity)

from sales

group by rollup(item_name, color, clothes_size);

group by rollup(item name, color, clothes size) generates only 4 groupings:

{ (item_name, color, clothes_size), (item_name, color), (item_name), () }

Multiple rollups and cubes can be used in a single group by clause. For instance, the following query:

SQL>select item_name, color, clothes_size, sum(quantity)

from sales group by rollup(item_name), rollup(color, clothes_size);

generates the groupings:

{ (item_name, color, clothes_size), (item_name, color), (item_name), (color, clothes_size), (color), () }

Formal Relational Query Languages

The Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, and assignment. We shall define these operations in terms of the fundamental operations.

Fundamental Operations:

The select, project, and rename operations are called unary operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called binary operations.

The Select Operation:

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ . The argument relation is in parentheses after the σ .56

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 6.1 The *instructor* relation.

Thus, to select those tuples of the instructor relation where the instructor is in the "Physics" department, we write:

$$\sigma_{\text{dept name = "Physics"}} \textbf{(instructor)}$$

ID	name	dept_name	salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 6.2 Result of $\sigma_{dept\_name = \text{"Physics"}}(instructor)$.

We can find all instructors with salary greater than $90,000 by writing:

$$\sigma_{\text{salary>90000}}(\text{instructor})$$

In general, we allow comparisons using =, =, , and ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (¬). Thus, to find the instructors in Physics with a salary greater than $90,000, we write:

$$\sigma_{\text{dept\_name = "Physics"} \wedge \text{salary>90000}}(\text{instructor})$$

The Project Operation:

Suppose we want to list all instructors' ID, name, and salary, but do not care about the dept name. The project operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Projection is denoted by the uppercase Greek letter pi (π). We list those attributes that we wish to appear in the result as a subscript to π. The argument relation follows in parentheses. We write the query to produce such a list as:57

$$\pi_{\text{ID, name, salary}}(\text{instructor})$$

ID	name	salary
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Figure 6.3 Result of $\Pi_{ID,\ name,\ salary}(instructor)$.

Composition of Relational Operations:

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find the name of all instructors in the Physics department." We write:

$$\pi_{\text{name}}(\sigma_{\text{dept\_name = "Physics"}}(\text{instructor}))$$

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a relational-algebra expression. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as +, −, ∗, and ÷) into arithmetic expressions.

The Union Operation:

Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both. The information is contained in the section relation.

$$\Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year=2009}(section))$$

To find the set of all courses taught in the Spring 2010 semester, we write:

$$\Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year=2010}(section))$$

To answer the query, we need the union of these two sets; that is, we need all section IDs that appear in either or both of the two relations.

by the binary operation union, denoted, as in set theory, by ∪. So the expression needed is:

$$\Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year=2010}(section))$$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 6.5 Courses offered in either Fall 2009, Spring 2010 or both semesters.

Therefore, for a union operation r ∪ s to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

The Set-Difference Operation:

The set-difference operation, denoted by −, allows us to find tuples that are in one relation but are not in another. The expression r − s produces a relation containing those tuples in r but not in s.

We can find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester by writing:

$$\Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year=2009}(section)) - \Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year=2010}(section))$$

course_id
CS-347
PHY-101

Figure 6.6 Courses offered in the Fall 2009 semester but not in Spring 2010 semester.

The Cartesian-Product Operation:

The Cartesian-product operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations r1 and r2 as r1 × r2.

For example, the relation schema for r = instructor × teaches is:

(instructor.ID, instructor.name, instructor.dept_name, instructor.salary
teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

With this schema, we can distinguish instructor.ID from teaches.ID. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as:

(instructor.ID, name, dept_name, salary

teaches.ID, course_id, sec_id, semester, year)

In general, if we have relations r1(R1) and r2(R2), then r1 × r2 is a relation whose schema is the concatenation of R1 and R2. Relation R contains all tuples t for which there is a tuple t1 in r1 and a tuple t2 in r2 for which t[R1] = t1[R1] and t[R2] = t2[R2].

Suppose that we want to find the names of all instructors in the Physics department together with the course id of all courses they taught. We need the information in both the instructor relation and the teaches relation to do so. If we write:

$$\sigma_{\text{dept\_name = "Physics"}}(\text{instructor} \times \text{teaches})$$

tuple in $\sigma_{\text{dept\_name = "Physics"}}(\text{instructor} \times \text{teaches})$ that contains his name, and which satisfies instructor.ID = teaches.ID. So, if we write:

$$\pi_{\text{instructor .ID = teaches.ID}} (\sigma_{\text{dept\_name = "Physics"}}(\text{instructor} \times \text{teaches}))$$

Finally, since we only want the names of all instructors in the Physics department together with the course id of all courses they taught, we do a projection:

$$\pi_{\text{name, course\_id}}(\sigma_{\text{instructor .ID = teaches.ID}} (\sigma_{\text{dept\_name = "Physics"}}(\text{instructor} \times \text{teaches})))$$

name	*course_id*
Einstein	PHY-101

Figure 6.10 Result of $\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)))$.

The Rename Operation:

The rename operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression ρ_x **(E)** returns the result of expression E under the name x.

A relation r by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name. A second form of the rename operation is as follows: Assume that a relational algebra expression E has arity n. Then, the expression $\rho_x(A1,A2,...,An)$ **(E)** returns the result of expression E under the name x, and with the attributes renamed to A1, A2,..., An.

Formal Definition of the Relational Algebra:

A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

A constant relation is written by listing its tuples within { },

for example { (22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000) }.

A general expression in the relational algebra is constructed out of smaller subexpressions. Let E1 and E2 be relational-algebra expressions. Then, the following are all relational-algebra expressions:

- E1 ∪ E2
- E1 − E2
- E1 × E2
- σ_P(E1), where P is a predicate on attributes in E1
- π_S(E1), where S is a list consisting of some of the attributes in E1
- ρ_x (E1), where x is the new name for the result of E160

Additional Relational-Algebra Operations:

The Set-Intersection Operation: The first additional relational-algebra operation that we shall define is set intersection (∩). Suppose that we wish to find the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters. Using set intersection, we can write

$$\pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year=2009}(section)) \cap \pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year=2010}(section))$$

Note that we can rewrite any relational-algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r \cap s = r - (r - s)$$

It is simply more convenient to write r ∩ s than to write r − (r − s).

The Natural-Join Operation:

The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the join symbol ⋈.

“Find the names of all instructors together with the course id of all courses they taught.” We express this query by using the natural join as follows:

$$\pi_{name,\ course\_id}(instructor \bowtie teaches)$$

The **theta join** operation is a variant of the natural-join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations r(R) and s(S), and let Θ be a predicate on attributes in the schema R ∪ S. The theta join operation $r \bowtie_\theta S$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta (r \times s)$$

UNIT-III The Entity-Relationship model

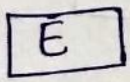
→ The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema & many database-design tools draw on concepts from the E-R model.

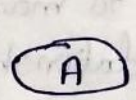
→ The E-R data model has 3 basic concepts
* entity sets
* Relationship sets
* Attributes.

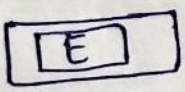
→ The E-R model also has an associated diagrammatic representation of the E-R diagram.

E-R Diagram Components :-

→ It is a pictorial representation of data that describes how data is communicated and related each other.


→ entity set


Attribute

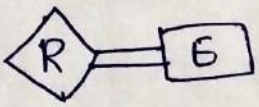

weak entity set

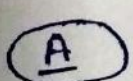

multivalued Attribute

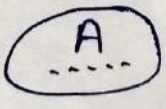

Relationship set

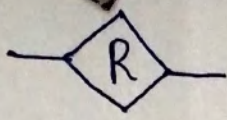

Derived attribute


Identifying Relationship set for weak entity set

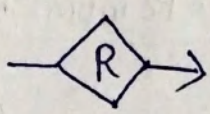

total participation of entity set in Relationship


primary key

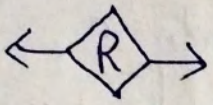

Descriminating Attribute of weak entity set



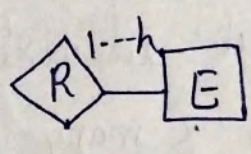
Many to many Relationship



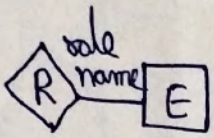
many to one Relationship



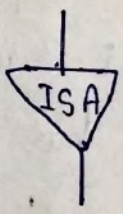
one to one Relationship



cardinality limits

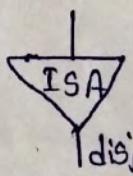


Role indication



ISA (specialization or generalization)

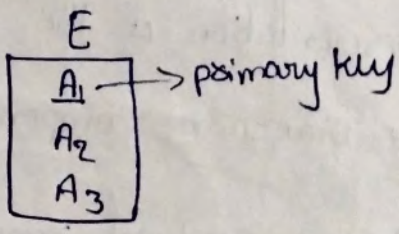
Total Generalization

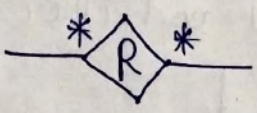


Disjoint Generalization.

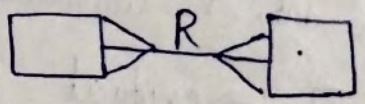
Alternative E-R notation :-

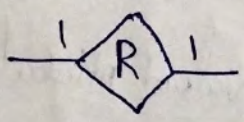
Entity set E with attributes A_1, A_2, A_3 and primary key A_1



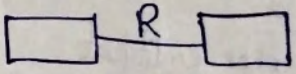


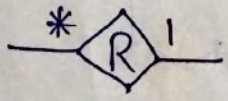
Many to many Relationship



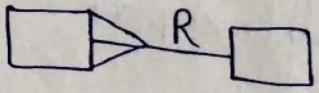


one to one Relationship

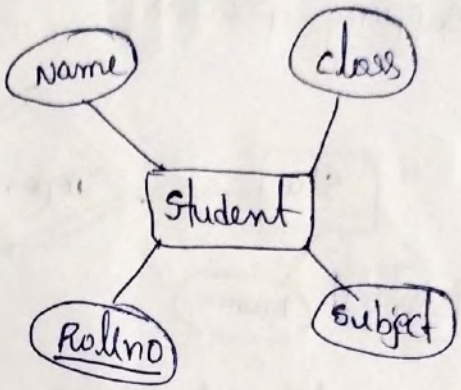




Many to one Relationship



Mapping Entity :-



Mapping process:

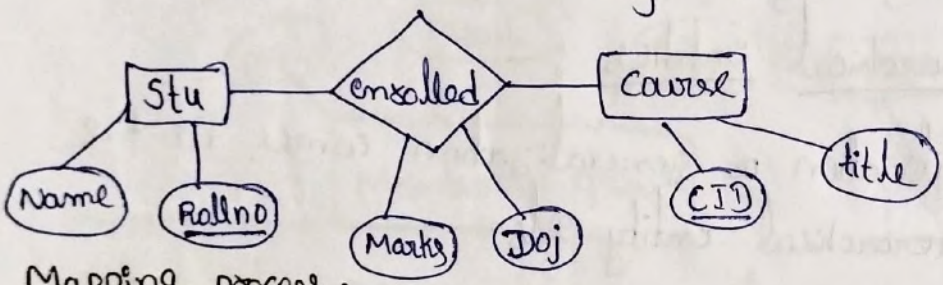
→ create table for each entity.

→ entities attribute should become fields of tables with respective datatypes.

→ Declare primary key.

Mapping Relationship :-

→ It is an association among entities.



Mapping process:

→ Create table for a relationship

→ Add primary key of all participanty entities as fields

→ If relationship has any attribute, then add each attribute as field of a table.

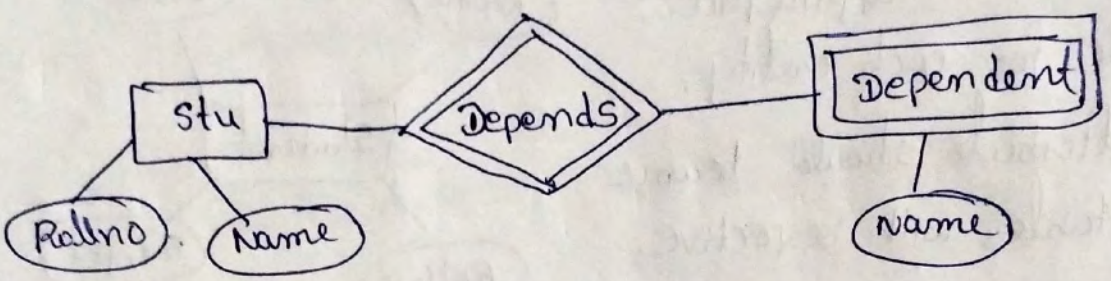
→ Declare primary key

→ Declare Foreign key constrainty.

Mapping weak entity set :-

→ A weak entity set is one which does not have any primary key associated with it.

Mapping process :



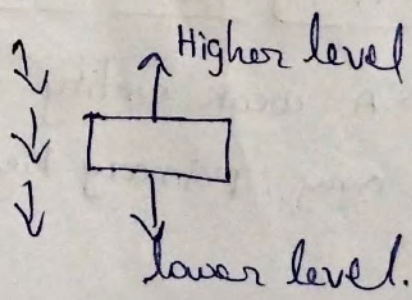
→ create table for weak entity set.
→ Add all attribute to table as fields
→ Add primary key of identifying entity set.
→ Declare Foreign key constraints.

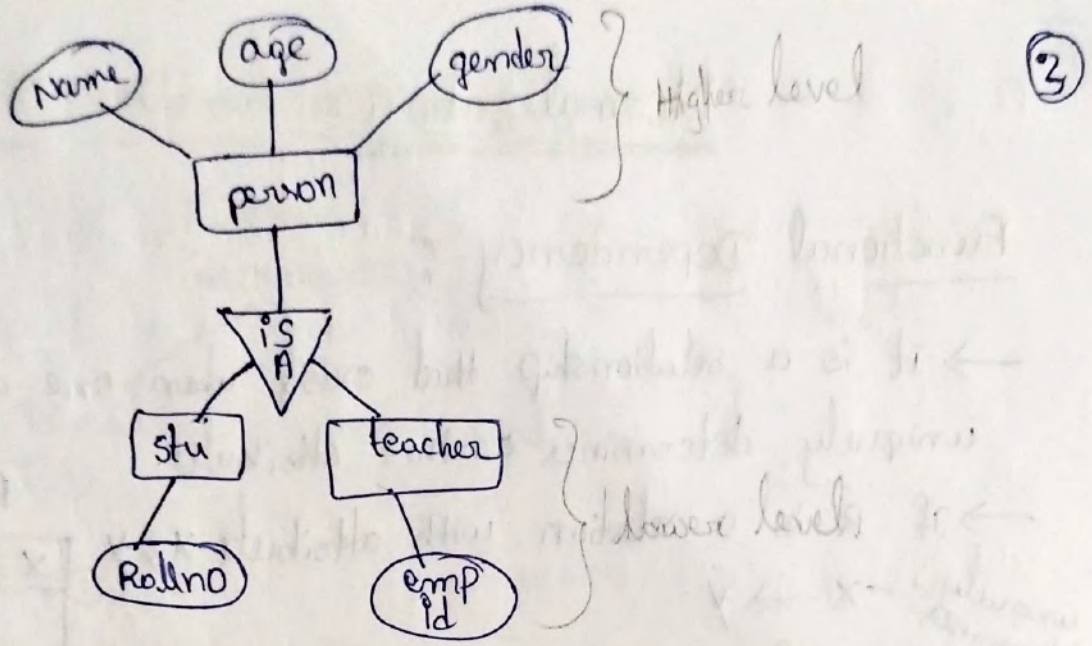
Mapping Hierarchical entities :—

→ ER specialization or generalization comes in the form of hierarchical entity sets.

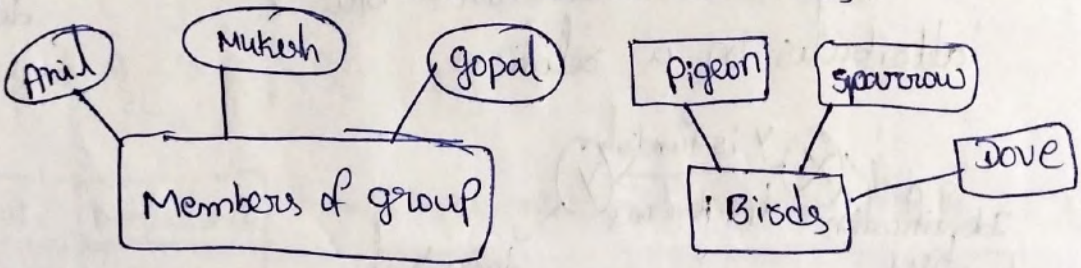
Mapping process :

→ create table for all higher entities & lower level entities.
→ Add PK of higher level in the table of lower level entities.
→ In lower-level ~~ent~~ tables, add all other attribute of lower level entities.
→ Declare P.K of higher level table & lower level table
→ Declare F.K constraints





Generalization :- no. of entities brought together into one entity based on their similar characteristics.



Specialization :- opposite to generalization.

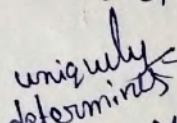
⟶ Group of entities divided into group of entities divided into subgroups based on their characteristics

Normalization

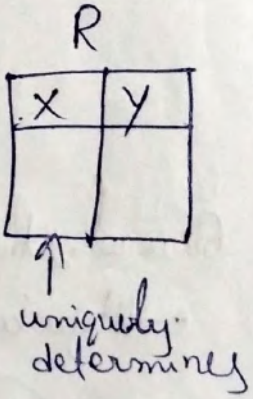
Functional Dependency :-

→ It is a relationship that exists when one attribute uniquely determines another attribute.

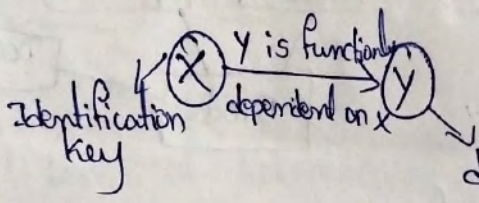
→ If R is a relation with attribute X & Y



Y is functionally dependent on X.



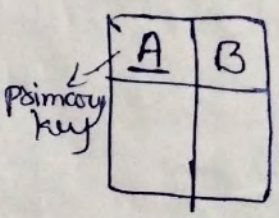
→ It is a set of constrains b/w 2 attributes in a relation.



Example :-

if every attribute B of R dependent of A, then attribute A is a primary key.

$A \rightarrow B$



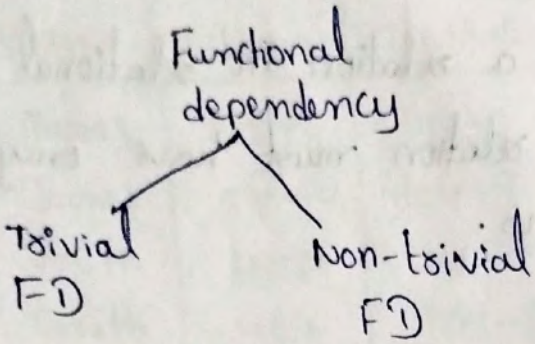
Ex :-

ID	name	surname
S_1	Bhanu	P
S_2	priya	G
S_3	Bhanu	M

identification key

ID → name
ID → surname
} functionally dependent

Types of Functional dependency:

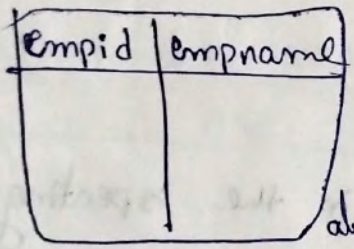


* Trivial FD :-

⟹ A → B has trivial FD if B is a subset of A.

→ The following dependencies are also trivial like : A → A, B → B.

Ex:-



empid, empname → empid is a TFD

empid is subset of {empid, empname}

also empid → empid & empname → empname are trivial dependencies too.

* Non-trivial FD :-

⟹ A → B has a non-trivial FD if B is not a subset of A.

→ when A intersection B is NULL, then A → B is called as complete non-trivial.

Ex:- id → name,

name → DOB.

First Normal Form (1NF) :-

→ It is a property of a relation in relational db.

→ All attribute in a relation must have only atomic (indivisible) domains.

Requirements :

→ Each table has primary key.

→ The value in each column of a table are atomic (no multi-value attribute allowed).

→ There are no repeating groups.
Two columns do not store similar info in the same table.

1NF Decomposition :

→ place all items that appear in the repeating group in a new table.

→ Designate a PK for each new table produce.

→ Duplicate in the new table the PK of the table from which the Repeating group was extracted or vice versa.

Un-normalized student table :

Sid	sname	sRoom	class1	class2
123	James	555	103-8	104-9
124	Smith	467	2040	102-8

Normalized student table :-

Sid	Sname	SRoom	class-#
123	James	555	103-8
123	James	555	104-9
124	Smith	467	209-0
124	Smith	467	102-8

Ex:-2

Un-Normalized Table

Course	content
Programming	Java, C++
web	HTML, PHP, ASP

Normalized table.

course	content
Programming	Java
Programming	C++
web	HTML
web	PHP
web	ASP

→ each attribute must contain only a single value from its predefined domain.

→ This INF used in most small to medium size applications.

2NF Normalization :- (1NF, Fully FD)

1) Prime attribute : An attribute which is a part of ~~PK~~ Candidate key

2) Non prime attribute : An attribute which is not a part of PK.

Requirements :-

1) The db is in 1NF
2) All non key attribute in table must be fully Functionly dependent on PK.

Definition :-

→ Every non-prime attribute should be fully FD on prime attribute.

if $X \longrightarrow A$ holds, then there should not be any proper subset Y of X, for which $Y \longrightarrow A$ also holds true.

~~→ If a data~~

2NF Decomposition :-

→ If a data item is fully FD on only a part of PK, move that data item & part of PK to a new table.

→ If other data item are FD on the same part of key, place them in a new table.

→ Make the partial PK copied from original table the PK for new table.

→ place all items that appear in the repeating group in new table.

Ex :-

Sid	Lastname	pafid	pafname	Grade
1	Tailor	3	sachin	5
2	Jhon	2	parker	4
3	Moni	1	James	6

Student

Sid	lastname
1	Tailor
2	Jhon
3	Moni

pafessor

pafid	Paf
1	James
2	parker
3	sachin

Grades

sid	pafid	Grade
1	3	5
2	2	4
3	1	6

↑ Here Grade is Fully FD on sid, pafid.

Ex :- 2

Schema → {city, street, HNO, Hcolor, citypopulation}

1) key → {city, street, HNO}

2) {city, street, HNO} → {Hcolor}

3) {city} → {city population} → non key

4) city population does not belong to any key

5) citypopulation is FD on city which is a proper subset of the key.

convert to 2NF :

old schema → {city, street, HNO, Hcolor, citypopulation}

① New schema → { city, street, HNO, Hcolor } → non key att
FFD

② New schema → { city, citypopulation }.
FFD

3NF Normalization :-

→ All non key attribute of a table must be FD on candidate key. (There can be no interdependencies among non key attribute).

→ It is in 2NF.

→ There is no transitive FD.

A → B
B → C
A → C via B.

→ 3NF is used to reduce the Redundant data.

Ex :-

Bookid	Genre id	Genre type	price
1	1	science	36
2	2	Maths	35
3	1	Science	25
4	3	physics	20
5	2	Maths	30

Bookid → Genreid
Genreid → Genretype
Bookid → Genretype via Genreid. } Transitive FD

↓
decomposing

Book

Bookid	Genreid	price
1	1	30
2	2	35
3	1	25
4	3	20
5	2	30

Genre

Genreid	Genretype
1	Science
2	maths
3	physics

BCNF Boyce-codd Normal form :-

→ BCNF is an advanced version of 3NF called as 3.5 NF.

→ For every FD $x \to y$, x should be superkey of table.

Ex :-

Stu	course	Teacher
pinky	DBMS	priya
Lucky	DBMS	madhu
Deepu	coa	bhanu
bunnu	coa	bhanu
Janu	DBMS	madhu

SK

key = {stu, course}

{stu, course} → Teacher

Candidate is not a ~~super~~ key.

Stu	course
pinky	DBMS
Lucky	DBMS
Deepu	coa
bunnu	coa
Janu	DBMS

SK

course	teacher
DBMS	priya
DBMS	madhu
coa	Bhanu

SK

4NF Normalization :- 2NF, 3NF, BCNF → based on key &

→ 4NF based on keys & multivalued dependencies.

→ 4NF eliminates independent many-to-one relationships b/w columns.

→ A relation must first be in BCNF

→ A given relation may not contain more than one multivalued attributes.

→ For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Ex :-

Stuid	subject	Activity
100	music	Swimming
100	Accounting	"
100	Music	Tennis
100	Accounting	Tennis
150	Maths	Jogging

PK ⇒ {stuid, subject, Activity}

→ many stuid have same subject

→ many stuid have same activity

→ Thus violates 4NF.

⇓ decompose to 4NF.

old schema → {stuid, sub, Activity}

1) new schema → {stuid, Sub}

2) new schema → {stuid, Activity}

Stuid	Subject
100	Music
100	Accounting
150	Maths

Stuid	Activity
100	Swimming
100	Tennis
150	Jogging

So, now there is no multivalued attribute.

5NF Normalization :-

→ A relation is in 5NF if it is in 4NF & not contains any join dependency & joining should be lossless. or lossless join decomposition.

→ 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.

→ 5NF is also known as project-join normal form (PJ/NF).

Ex :-

Subject	Teacher	Semester
Computer	Anu	sem 1
Computer	John	sem 1
Maths	John	sem 1
Math	Akash	sem 2
Chemistry	Praveen	Sem 1

$\Rightarrow$ P_1

Semester	Subject
sem 1	computer
sem 1	Maths
sem 1	chemistry
sem 2	Maths

P_2

Subject	teacher
Computer	Anu
Computer	John
Maths	John
Maths	Akash
Chemistry	praveen

P_3

Semester	teacher
Sem1	Anu
Sem1	John
sem1	John
Sem2	Akash
Sem1	praveen

Ex :-

Agent	company	productname
Suneet	ABC	Nut
Raj	ABC	Bolt
Raj	ABC	Nut
Suneet	CDE	Bolt
Suneet	ABC	Bolt

$\Rightarrow$

P_1

Agent	company
Suneet	ABC
Suneet	CDE
Raj	ABC

P_2

Agent	pname
Suneet	Nut
Suneet	Bolt
Raj	Bolt
Raj	Nut

P_3

Company	pname
ABC	Nut
ABC	Bolt
CDE	Bolt

P_3

$\rightarrow$ perform natural join of P_1 & P_2, All redundancy has been removed, if the natural join P_1 & P_2 is taken, the result is:

Agent	company	pname
Suneet	ABC	Nut
Suneet	ABC	Bolt
Suneet	CDE	Nut
Suneet	CDE	Bolt
Raj	ABC	Bolt
Raj	ABC	Nut

→ now if this result is joined with P_3 over the column 'company' & 'product_name' the following table is obtained:

⇓

Agent	Company	product_name
Suneet	ABC	Nut
Suneet	ABC	Bolt
Suneet	CDE	Bolt
Raj	ABC	Bolt
Raj	ABC	Nut

Ex:-

empid	ename	eage	ecity	Did	dname
22	abc	28	Mumbai	827	sales
33	Alina	25	Delhi	438	Marketing
46	stephan	30	Banglore	869	Finance
52	kasthi	36	Mumbai	575	production
60	Jack	40	Noida	678	Testing

decompose into two relations emp & dept.

emp

empid	ename	eage	ecity
22	ABC	28	Mumbai
33	Alina	25	Delhi
46	Stephan	30	Bangalore
52	Karthi	36	Mumbai
60	Jack	40	Noida

Dept

Did	empid	Dname
827	22	Sales
438	33	Marketing
869	46	Finance
575	52	Production
678	60	Testing

→ natural join on emp & Dept.

empid	ename	eage	ecity	did	Dname
22	ABC	28	Mumbai	827	Sales
33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Karthi	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

→ Hence, the decomposition is lossless join decomposition.

Multivalued Dependency :-

→ Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.

→ A multivalued dependency consists of at least two attributes that are dependent on a 3rd attribute so, it always requires at least 3 attributes.

Ex :-

Bike model	Manuf-yr	color
M2011	2008	white
M2001	2008	Black
M3001	2013	white
M3001	2013	Black
M4006	2017	white
M4006	2017	Black.

→ Here columns color & Manuf-yr are dependent on Bike-model & independent of each other.

→ these two columns can be called as multivalued dependent on Bike-model. The representation of these dependencies is shown below:

Bike model →→ manuf-yr

Bike-model →→ color.

→ This can be read as "Bike-model multidetermined manuf-yr & Bike-model multidetermined color.

UNIT- 4

Query Processing: Overview, Measures of Query cost, Selection operation, sorting, Join Operation, other operations, Evaluation of Expressions.

Query optimization: Overview, Transformation of Relational Expressions, Estimating statistics of Expression results, Choice of Evaluation Plans, Materialized views, Advanced Topics in Query Optimization.

$\pi \quad \sigma_{salary<75000} (\pi_{salary}(instructor))$

$\pi_{salary}(\sigma_{salary<75000} (instructor))$

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

Overview

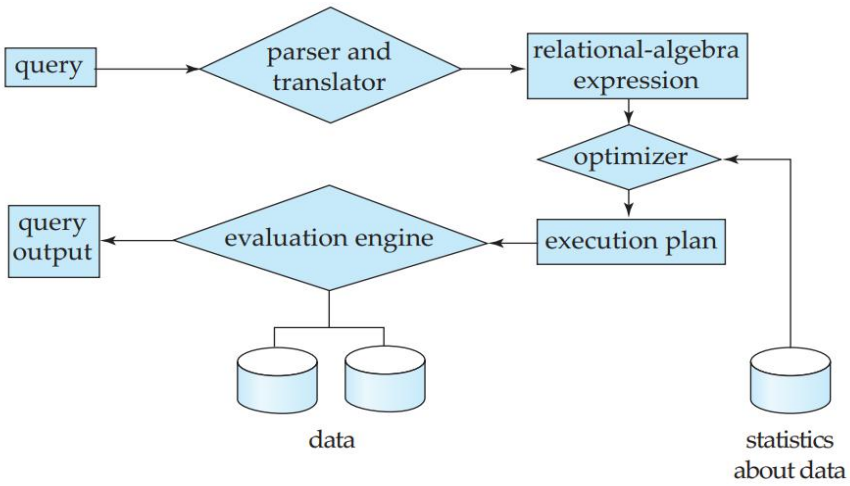


Figure 12.1 Steps in query processing.

The steps involved in processing a query appear in Figure 12.1. The basic steps are:

1. Parsing and translation.

2. Optimization.

3. Evaluation.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the

query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

consider the query:

SQL> select salary from instructor where salary < 75000;

This query can be translated into either of the following relational-algebra expressions:

$\sigma_{salary<75000}(\pi_{salary}(instructor))$

$\pi_{salary}(\sigma_{salary<75000}(instructor))$

A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive. A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or query-evaluation plan. Figure 12.2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as “index 1”) is specified for the selection operation. The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

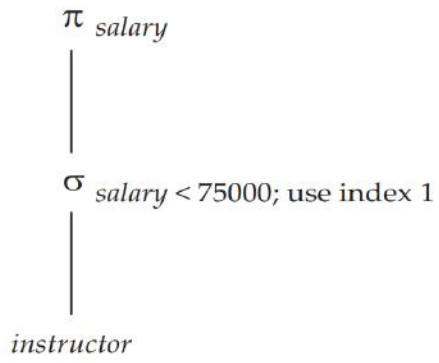


Figure 12.2 A query-evaluation plan.

It is the responsibility of the system to construct a queryevaluation plan that minimizes the cost of query evaluation; this task is called **query optimization**.

Measures of Query Cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.

We use the number of block transfers from disk and the number of disk seeks to estimate the cost of a query-evaluation plan. If the disk subsystem takes an average of t_T seconds to transfer a block of data, and has an average block-access time (disk seek time plus rotational latency) of t_S seconds, then an operation that transfers b blocks and performs S seeks would take $b * t_T + S * t_S$ seconds. The values of t_T and t_S must be calibrated for the disk system used, but typical values for high-end disks today would be t_S = 4 milliseconds and t_T = 0.1 milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.

The response time for a query-evaluation plan (that is, the wall-clock time required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan. Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following reasons:

1. The response time depends on the contents of the buffer when the query begins execution; this information is not available when the query is optimized, and is hard to account for even if it were available.

2. In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

As a result, instead of trying to minimize the response time, optimizers generally try to minimize the total resource consumption of a query plan. Our model of estimating the total disk access time (including seek and data transfer) is an example of such a resource consumption–based model of query cost.

Selection Operation

In query processing, the file scan is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

Selections Using File Scans and Indices:

Consider a selection operation on a relation whose tuples are stored together in one file. The most straightforward way of performing a selection is as follows:

• **A1 (linear search).** In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file. In case blocks of the file are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

Search algorithms that use an index are referred to as index scans. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

• **A2 (primary index, equality on key).** For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.

• **A3 (primary index, equality on nonkey).** We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, A.

The only difference from the previous case is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key.

• **A4 (secondary index, equality).** Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the indexing field is not a key.

In the first case, only one record is retrieved. The time cost in this case is the same as that for a primary index (case A2).

In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer. The worst-case time cost in this case is $(h_i + n) * (t_S + t_T)$, where n is the number of records fetched, if each record is in a different disk block, and the block fetches are randomly ordered.

In certain algorithms, including A2, the use of a B+ -tree file organization can save one access since records are stored at the leaf-level of the tree.

Selections Involving Comparisons:

Consider a selection of the form $\sigma_{A \leq V}(r)$. We can implement the selection either by using linear search or by using indices in one of the following ways:4

• **A5 (primary index, comparison).** A primary ordered index (for example, a primary B+-tree index) can be used when the selection condition is a comparison. For comparison conditions of the form A > v or A ≥ v, a primary index on Acan be used to direct the retrieval of tuples, as follows: For A ≥ v, we look up the value v in the index to find the first tuple in the file that has a value of A = v. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For A > v, the file scan starts with the first tuple such that A > v. The cost estimate for this case is identical to that for case A3.

• **A6 (secondary index, comparison).** We can use a secondary ordered index to guide retrieval for comparison conditions involving . The lowest-level index blocks are scanned, either from the smallest value up to v (for < and ≤), or from v up to the maximum value (for > and ≥).

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2	Primary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Primary B^+-tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a primary index) and don't require additional seeks.
A4	Secondary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to primary index.
A4	Secondary B^+-tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Primary B^+-tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6	Secondary B^+-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Figure 12.3 Cost estimates for selection algorithms.

Implementation of Complex Selections:

So far, we have considered only simple selection conditions of the form A op B, where op is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction:** A *conjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction:** A *disjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions Θ_i .6

• **Negation:** The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_{\theta}(r)$.6

• **A7 (conjunctive selection using one index).** We first determine whether an access path is available for an attribute in one of the simple conditions. If one is, one of the selection algorithms A2 through A6 can retrieve records satisfying that condition. We complete the operation by testing, in the memory buffer, whether or not each retrieved record satisfies the remaining simple conditions.

• **A8 (conjunctive selection using composite index).** An appropriate composite index (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms A2, A3, or A4 will be used.

• **A9 (conjunctive selection by intersection of identifiers).** Another alternative for implementing conjunctive selection operations involves the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions.

The cost of algorithm A9 is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order.

• **A10 (disjunctive selection by union of identifiers).** If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records.

Sorting

Sorting of data plays an important role in database systems for two reasons. First, SQL queries can specify that the output be sorted. Second, and equally important for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted.

External Sort-Merge Algorithm:

Sorting of relations that do not fit in memory is called external sorting. The most commonly used technique for external sorting is the external sort–merge algorithm. We describe the external sort–merge algorithm next. Let M denote the number of blocks in the main-memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

1. In the first stage, a number of sorted runs are created; each run is sorted, but contains only some of the records of the relation.

```
i = 0;
repeat
    read M blocks of the relation, or the rest of the relation,
        whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file R_i;
    i = i + 1;
until the end of the relation
```

2. In the second stage, the runs are merged. Suppose, for now, that the total number of runs, N, is less than M, so that we can allocate one block to each run and have space left to hold one block of output. The merge stage operates as follows:

```
read one block of each of the N files R_i into a buffer block in memory;
repeat
    choose the first tuple (in sort order) among all buffer blocks;
    write the tuple to the output, and delete it from the buffer block;
    if the buffer block of any run R_i is empty and not end-of-file(R_i)
        then read the next block of R_i into the buffer block;
until all input buffer blocks are empty
```

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort– merge algorithm; it merges N runs, so it is called an N-way merge.

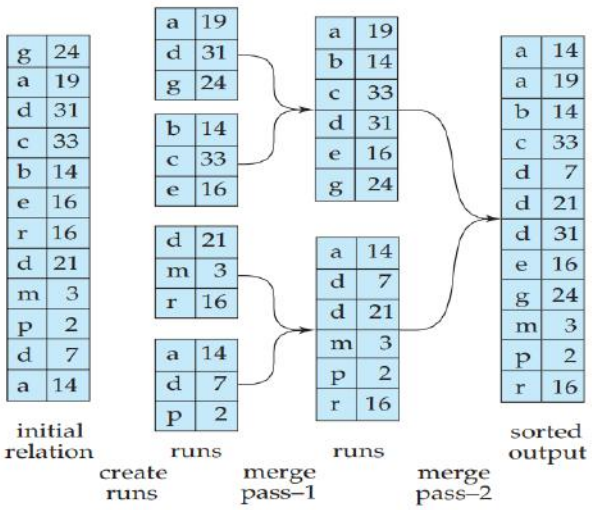


Figure 12.4 External sorting using sort–merge.

In this case, the merge operation proceeds in multiple passes. Since there is enough memory for M − 1 input buffer blocks, each merge can take M − 1 runs as input.

The initial pass functions in this way: It merges the first M − 1 runs (as described in item 2 above) to get a single run for the next pass. Then, it merges the next M − 1 runs similarly, and so on, until it has processed all the initial runs. At this point, the number of runs has been reduced by a factor of M − 1. If this reduced number of runs is still greater than or equal to M,

another pass is made, with the runs created by the first pass as input. Each pass reduces the number of runs by a factor of M − 1. The passes repeat as many times as required, until the number of runs is less than M; a final pass then generates the sorted output.

Cost Analysis of External Sort-Merge:

We compute the disk-access cost for the external sort–merge in this way: Let b_r denote the number of blocks containing records of relation r. The first stage reads every block of the relation and writes them out again, giving a total of 2br block transfers. The initial number of runs is $[\log_{M-1} br/M]$. Since the number of runs decreases by a factor of M − 1 in each merge pass, the total number of merge passes required is $[\log_{M-1}(br/M)]$. Each of these passes reads every block of the relation once and writes it out once, with two exceptions. First, the final pass can produce the sorted output without writing its result to disk. Second, there may be runs that are not read in or written out during a pass— for example, if there are M runs to be merged in a pass, M − 1 are read in and merged, and one run is not accessed during the pass. Ignoring the (relatively small) savings due to the latter effect, the total number of block transfers for external sorting of the relation is:

$$b_r(2[\log_{M-1}(br/M)]+1$$

Applying this equation to the example in above Figure 12.4, we get a total of 12 ∗ (4 + 1) = 60 block transfers.

The total number of seeks is then:

$$2\lceil b_r/M\rceil + \lceil b_r/b_b\rceil(2\lceil\log_{M-1}(b_r/M)\rceil - 1)$$

Applying this equation to the example in above Figure 12.4, we get a total of 8 + 12 ∗ (2 ∗ 2 − 1) = 44 disk seeks if we set the number of buffer blocks per run, b_b to 1.

Join Operation

In this section, we study several algorithms for computing the join of relations, and we analyze their respective costs.

We use the term equi-join to refer to a join of the $r \bowtie_{r.A=s.B} s$ form

where A and B are attributes or sets of attributes of relations r and s, respectively. We use as a running example the expression:

$$\text{student} \bowtie \text{takes}$$

We assume the following information about the two relations:

- Number of records of *student*: $n_{student} = 5,000$.
- Number of blocks of *student*: $b_{student} = 100$.
- Number of records of *takes*: $n_{takes} = 10,000$.
- Number of blocks of *takes*: $b_{takes} = 400$.

Nested-Loop Join:

Figure 12.5 shows a simple algorithm to compute the theta join, r ⋈θ s, of two relations r and s. This algorithm is called the nested-loop join algorithm, since it basically consists of a pair of nested for loops. Relation r is called the outer relation and relation s the inner relation of the join, since the loop for r encloses the loop for s. The algorithm uses the notation $t_r \cdot t_s$, where tr and ts are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s.9

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Figure 12.5 Nested-loop join.

Now consider the natural join of student and takes. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that student is the outer relation and takes is the inner relation in the join. We will have to examine 5000 ∗ 10, 000 = 50 ∗ 106 pairs of tuples. In the worst case, the number of block transfers is 5000 ∗ 400+100 = 2,000,100, plus 5000+100 = 5100 seeks. In the best case scenario, however, we can read both relations only once, and perform the computation. This computation requires at most 100 + 400 = 500 block transfers, plus 2 seeks—a significant improvement over the worst-case scenario. If we had used takes as the relation for the outer loop and student for the inner loop, the worst-case cost of our final strategy would have been 10,000 ∗ 100 + 400 = 1,000,400 block transfers, plus 10,400 disk seeks. The number of block transfers is significantly less, and although the number of seeks is higher, the overall cost is reduced, assuming t_S = 4 milliseconds and t_T = 0.1 milliseconds.

Block Nested-Loop Join:

block nested-loop join, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation. Thus, in the worst case, there will be a total of br ∗ bs + br block transfers, where br and bs denote the number of blocks containing records of r and s, respectively. Each scan of the inner relation requires one seek, and the scan of the outer relation requires one seek per block, leading to a total of 2 ∗ br seeks.

In the worst case, we have to read each block of takes once for each block of student. Thus, in the worst case, a total of 100 ∗ 400 + 100 = 40,100 block transfers plus 2∗100 = 200 seeks are required. This cost is a significant improvement over the 5000∗400+100 = 2,000,100 block

transfers plus 5100 seeks needed in the worst case for the basic nested-loop join. The best-case cost remains the same—namely, 100 + 400 = 500 block transfers and 2 seeks.

Indexed Nested-Loop Join:

In a nested-loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r.

This join method is called an indexed nested-loop join; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join. Looking up tuples in s that will satisfy the join conditions with a given tuple t_r is essentially a selection on s. For example, consider student ⋈ takes. Suppose that we have a student tuple with ID "00128". Then, the relevant tuples in takes are those that satisfy the selection "ID = 00128".

For example, consider an indexed nested-loop join of student ⋈ takes, with student as the outer relation. Suppose also that takes has a primary B+-tree index on the join attribute ID, which contains 20 entries on average in each index node. Since takes has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data. Since $n_{student}$ is 5000, the total cost is 100+5000 ∗ 5 = 25,100 disk accesses, each of which requires a seek and a block transfer. In contrast, as we saw before, 40,100 block transfers plus 200 seeks were needed for a block nested loop join.

Merge Join:

The merge-join algorithm (also called the sort-merge-join algorithm) can be used to compute natural joins and equi-joins. Let r(R) and s(S) be the relations whose natural join is to be computed, and let R ∩ S denote their common attributes. Suppose that both relations are sorted on the attributes R ∩ S. Then, their join can be computed by a process much like the merge stage in the merge –sort algorithm.

- **Merge-Join Algorithm:**

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
   begin
      t_s := tuple to which ps points;
      S_s := {t_s};
      set ps to point to next tuple of s;
      done := false;
      while (not done and ps ≠ null) do
         begin
            t_s' := tuple to which ps points;
            if (t_s'[JoinAttrs] = t_s[JoinAttrs])
               then begin
                        S_s := S_s ∪ {t_s'};
                        set ps to point to next tuple of s;
                    end
               else done := true;
         end
```

```
t_r := tuple to which pr points;
while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
    begin
        set pr to point to next tuple of r;
        t_r := tuple to which pr points;
    end
while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
    begin
        for each t_s in S_s do
            begin
                add t_s ⋈ t_r to result;
            end
        set pr to point to next tuple of r;
        t_r := tuple to which pr points;
    end
end.
```

Figure 12.7 Merge join.

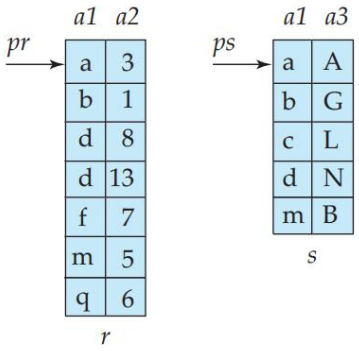


Figure 12.8 Sorted relations for merge join.

Figure 12.8 shows two relations that are sorted on their join attribute a1. It is instructive to go through the steps of the merge-join algorithm on the relations shown in the figure.

- **Cost Analysis:**

Suppose the relations are not sorted, and the memory size is the worst case, only three blocks. The cost is as follows:

1. Using the formulae that we developed in Section 12.4, we can see that sorting relation *takes* requires $\lceil \log_{3-1}(400/3) \rceil = 8$ merge passes. Sorting of relation *takes* then takes $400 * (2\lceil \log_{3-1}(400/3) \rceil + 1)$, or 6800, block transfers, with 400 more transfers to write out the result. The number of seeks required is $2 * \lceil 400/3 \rceil + 400 * (2 * 8 - 1)$ or 6268 seeks for sorting, and 400 seeks for writing the output, for a total of 6668 seeks, since only one buffer block is available for each run.
2. Similarly, sorting relation *student* takes $\lceil \log_{3-1}(100/3) \rceil = 6$ merge passes and $100 * (2\lceil \log_{3-1}(100/3) \rceil + 1)$, or 1300, block transfers, with 100 more transfers to write it out. The number of seeks required for sorting *student* is $2 * \lceil 100/3 \rceil + 100 * (2 * 6 - 1) = 1164$, and 100 seeks are required for writing the output, for a total of 1264 seeks.
3. Finally, merging the two relations takes $400 + 100 = 500$ block transfers and 500 seeks.

Thus, the total cost is 9100 block transfers plus 8932 seeks if the relations are not sorted, and the memory size is just 3 blocks.

With a memory size of 25 blocks, and the relations not sorted, the cost of sorting followed by merge join would be as follows:

1. Sorting the relation *takes* can be done with just one merge step, and takes a total of just $400 * (2\lceil \log_{24}(400/25) \rceil + 1) = 1200$ block transfers. Similarly, sorting *student* takes 300 block transfers. Writing the sorted output to disk requires $400 + 100 = 500$ block transfers, and the merge step requires 500 block transfers to read the data back. Adding up these costs gives a total cost of 2500 block transfers.
2. If we assume that only one buffer block is allocated for each run, the number of seeks required in this case is $2 * \lceil 400/25 \rceil + 400 + 400 = 832$ seeks for sorting *takes* and writing the sorted output to disk, and similarly $2 * \lceil 100/25 \rceil + 100 + 100 = 208$ for *student*, plus $400 + 100$ seeks for reading the sorted data in the merge-join step. Adding up these costs gives a total cost of 1640 seeks.

- **Hybrid Merge Join:**

To avoid this cost, we can use a hybrid merge-join technique that combines indices with merge join. Suppose that one of the relations is sorted; the other is unsorted, but has a secondary B+-tree index on the join attributes. The hybrid merge-join algorithm merges the sorted relation with the leaf entries of the secondary B+-tree index. The result file contains tuples from the sorted relation and addresses for tuples of the unsorted relation. The result file is then sorted on the addresses of tuples of the unsorted relation, allowing efficient retrieval of the corresponding tuples, in physical storage order, to complete the join.

- **Hash Join:**

Like the merge-join algorithm, the hash-join algorithm can be used to implement natural joins and equi-joins. In the hash-join algorithm, a hash function h is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have the same hash value on the join attributes. We assume that:

- h is a hash function mapping *JoinAttrs* values to $\{0, 1, \ldots, n_h\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
- $r_0, r_1, \ldots, r_{n_h}$ denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition r_i, where $i = h(t_r[JoinAttrs])$.
- $s_0, s_1, \ldots, s_{n_h}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.

- **Basics:**

For example, if d is a tuple in student, c a tuple in takes, and h a hash function on the ID attributes of the tuples, then d and c must be tested only if h(c) = h(d). If h(c) = h(d), then c and d must have different values for ID. However, if h(c) ≠ h(d), we must test c and d to see whether the values in their join attributes are the same, since it is possible that c and d have different iids that have the same hash value.

```
/* Partition s */
for each tuple t_s in s do begin
   i := h(t_s[JoinAttrs]);
   H_si := H_si ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
   i := h(t_r[JoinAttrs]);
   H_ri := H_ri ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
   read H_si and build an in-memory hash index on it;
   for each tuple t_r in H_ri do begin
      probe the hash index on H_si to locate all tuples t_s
         such that t_s[JoinAttrs] = t_r[JoinAttrs];
      for each matching tuple t_s in H_si do begin
         add t_r ⋈ t_s to the result;
      end
   end
end
```

Figure 12.10 Hash join.

- **Recursive Partitioning:**

The hash function used in a pass is, of course, different from the one used in the previous pass. The system repeats this splitting of the input until each partition of the build input fits in memory. Such partitioning is called recursive partitioning.

- **Handling of Overflows:**

Hash-table overflow occurs in partition i of the build relation s if the hash index on s_i is larger than main memory. Hash-table overflow can occur if there are many tuples in the build relation with the same values for the join attributes, or if the hash function does not have the properties of randomness and uniformity. In either case, some of the partitions will have more tuples than the average, whereas others will have fewer; partitioning is then said to be skewed.

The number of partitions is therefore increased by a small value, called the fudge factor.

Hash-table overflows can be handled by either overflow resolution or overflow avoidance. Overflow resolution is performed during the build phase, if a hash-index overflow is detected. Overflow resolution proceeds in this way: If s_i , for any i, is found to be too large, it is further partitioned into smaller partitions by using a different hash function. Similarly, r_i is also partitioned using the new hash function, and only tuples in the matching partitions need to be joined.

In contrast, overflow avoidance performs the partitioning carefully, so that overflows never occur during the build phase. In overflow avoidance, the build relation s is initially partitioned into many small partitions, and then some partitions are combined in such a way that each combined partition fits in memory. The probe relation r is partitioned in the same way as the combined partitions on s, but the sizes of r_i do not matter.

- **Cost of Hash Join:**

a hash join is estimated to require:

$$3(b_r + b_s) + 4n_h$$

block transfers. The overhead $4n_h$ is usually quite small compared to $b_r + b_s$, and can be ignored.

- Assuming b_b blocks are allocated for the input buffer and each output buffer, partitioning requires a total of $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ seeks. The build and probe phases require only one seek for each of the n_h partitions of each relation, since each partition can be read sequentially. The hash join thus requires $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$ seeks.

Now consider the case where recursive partitioning is required. Each pass reduces the size of each of the partitions by an expected factor of $M - 1$; and passes are repeated until each partition is of size at most M blocks. The expected number of passes required for partitioning s is therefore $\lceil \log_{M-1}(b_s) - 1 \rceil$.

- Since, in each pass, every block of s is read in and written out, the total block transfers for partitioning of s is $2b_s \lceil \log_{M-1}(b_s) - 1 \rceil$. The number of passes for partitioning of r is the same as the number of passes for partitioning of s, therefore the join is estimated to require:

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

block transfers.

- Again assuming b_b blocks are allocated for buffering each partition, and ignoring the relatively small number of seeks during the build and probe phase, hash join with recursive partitioning requires:

$$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil \log_{M-1}(b_s) - 1 \rceil$$

disk seeks.

- **Hybrid Hash Join:**

The **hybrid hash-join** algorithm performs another optimization; it is useful when memory sizes are relatively large, but not all of the build relation fits in memory. The partitioning phase of the hash-join algorithm needs a minimum of one block of memory as a buffer for each partition that is created, and one block of memory as an input buffer. To reduce the impact of seeks, a larger number of blocks would be used as a buffer; let b_b denote the number of blocks used as a buffer for the input and for each partition. Hence, a total of $(n_h + 1) * b_b$ blocks of memory are needed for partitioning the two relations. If memory is larger than $(n_h + 1) * b_b$, we can use the rest of memory ($M - (n_h + 1) * b_b$ blocks) to buffer the first partition of the build input (that is, s_0), so that it will not need to be written out and read back in. Further, the hash function is designed in such a way that the hash index on s_0 fits in $M - (n_h + 1) * b_b$ blocks, in order that, at the end of partitioning of s, s_0 is completely in memory and a hash index can be built on s_0.

- **Complex Joins:**

Consider the following join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

One or more of the join techniques described earlier may be applicable for joins on the individual conditions $r \bowtie_{\theta_1} s$, $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$, and so on. We can compute the overall join by first computing the result of one of these simpler joins $r \bowtie_{\theta_i} s$; each pair of tuples in the intermediate result consists of one tuple from r and one from s. The result of the complete join consists of those tuples in the intermediate result that satisfy the remaining conditions:

$$\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$$

These conditions can be tested as tuples in $r \bowtie_{\theta_i} s$ are being generated.

A join whose condition is disjunctive can be computed in this way. Consider:

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

The join can be computed as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

Other Operations

Duplicate Elimination:

We can implement duplicate elimination easily by sorting. Identical tuples will appear adjacent to each other as a result of sorting, and all but one copy can be removed. With external sort–merge, duplicates found while a run is being created can be removed before the run is written to disk, thereby reducing the number of block transfers. The remaining duplicates can be eliminated during merging, and the final sorted run has no duplicates. The worst-case cost estimate for duplicate elimination is the same as the worst-case cost estimate for sorting of the relation. We can also implement duplicate elimination by hashing, as in the hash-join algorithm.

Projection:

We can implement projection easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records. If the attributes in the projection list include a key of the relation, no duplicates will exist; hence, duplicate elimination is not required. Generalized projection can be implemented in the same way as projection.

Set Operations:

Hashing provides another way to implement these set operations. The first step in each case is to partition the two relations by the same hash function, and thereby create the partitions

$r_0, r_1, ..., r_{nh}$ and $s_0, s_1, ..., s_{nh}$. Depending on the operation, the system then takes these steps on each partition i = 0, 1,..., n_h:

• **r U s**

1. Build an in-memory hash index on r_i .

2. Add the tuples in si to the hash index only if they are not already present.

3. Add the tuples in the hash index to the result.

• **r ∩ s**

1. Build an in-memory hash index on r_i .

2. For each tuple in s_i , probe the hash index and output the tuple to the result only if it is already present in the hash index.

• **r – s**

1. Build an in-memory hash index on r_i .

2. For each tuple in s_i , probe the hash index, and, if the tuple is present in the hash index, delete it from the hash index.

3. Add the tuples remaining in the hash index to the result.

Outer Join:

Recall the *outer-join operations* described in Section 4.1.2. For example, the natural left outer join $takes \supset\!\bowtie student$ contains the join of *takes* and *student*, and, in addition, for each *takes* tuple t that has no matching tuple in *student* (that is, where *ID* is not in *student*), the following tuple t_1 is added to the result. For all attributes in the schema of *takes*, tuple t_1 has the same values as tuple t. The remaining attributes (from the schema of *student*) of tuple t_1 contain the value null.

We can implement the outer-join operations by using one of two strategies:

1. Compute the corresponding join, and then add further tuples to the join result to get the outer-join result. Consider the left outer-join operation and two relations: $r(R)$ and $s(S)$. To evaluate $r \supset\!\bowtie_\theta s$, we first compute $r \bowtie_\theta s$, and save that result as temporary relation q_1. Next, we compute $r - \Pi_R(q_1)$ to obtain those tuples in r that do not participate in the theta join. We can use any of the algorithms for computing the joins, projection, and set difference described earlier to compute the outer joins. We pad each of these tuples with null values for attributes from s, and add it to q_1 to get the result of the outer join.

   The right outer-join operation $r \bowtie\!\subset_\theta s$ is equivalent to $s \supset\!\bowtie_\theta r$, and can therefore be implemented in a symmetric fashion to the left outer join. We can implement the full outer-join operation $r \supset\!\bowtie\!\subset_\theta s$ by computing the join $r \bowtie s$, and then adding the extra tuples of both the left and right outer-join operations, as before.

2. Modify the join algorithms. It is easy to extend the nested-loop join algorithms to compute the left outer join: Tuples in the outer relation that do not match any tuple in the inner relation are written to the output after being padded with null values. However, it is hard to extend the nested-loop join to compute the full outer join.

Aggregation:

Recall the aggregation function (operator). For example, the function

SQL> select dept_name, avg (salary) from instructor group by dept_name;

computes the average salary in each university department.

The cost estimate for implementing the aggregation operation is the same as the cost of duplicate elimination, for aggregate functions such as min, max, sum, count, and avg.

Evaluation of Expressions

We consider both the materialization approach and the pipelining approach.

Materialization:

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an operator tree. Consider the expression:

$$\Pi_{name}(\sigma_{building = \text{“Watson”}}(department) \bowtie instructor)$$

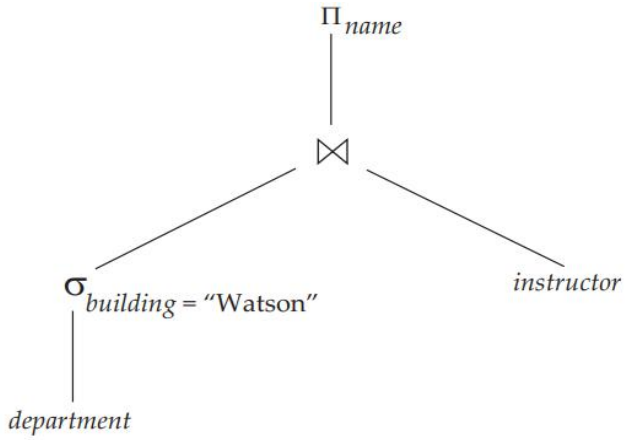


Figure 12.11 Pictorial representation of an expression.

If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation: the selection operation on department. The inputs to the lowest-level operations are relations in the database. We execute these operations by the algorithms that we studied earlier, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our example, the inputs to the join are the instructor relation and the temporary relation created by the selection on department. The join can now be evaluated, creating another temporary relation.

Double buffering (using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU

activity in parallel with I/O activity. The number of seeks can be reduced by allocating extra blocks to the output buffer, and writing out multiple blocks at once.

Pipelining:

We can improve query-evaluation efficiency by reducing the number of temporary files that are produced. We achieve this reduction by combining several relational operations into a pipeline of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called pipelined evaluation.

Creating a pipeline of operations can provide two benefits:

1. It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.

2. It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

- ### Implementation of Pipelining:

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for some frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline.

Pipelines can be executed in either of two ways:

1. In a demand-driven pipeline, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.

2. In a producer-driven pipeline, operations do not wait for requests to produce tuples, but instead generate the tuples eagerly. Each operation in a producer-driven pipeline is modeled as a separate process or thread within the system that takes a stream of tuples from its pipelined inputs and generates a stream of tuples for its output.

- ### Evaluation Algorithms for Pipelining:

The indexed nested loops join algorithm can output result tuples as it gets tuples for the outer relation. It is therefore said to be pipelined on its outer (left-hand side) relation, although it is blocking on its indexed (right-hand side) input, since the index must be fully constructed before the indexed nested-loop join algorithm can execute.

Hybrid hash join can be viewed as partially pipelined on the probe relation, since it can output tuples from the first partition as tuples are received for the probe relation. However, tuples

that are not in the first partition will be output only after the entire pipelined input relation is received.

The double-pipelined join algorithm in Figure 12.12 assumes that both inputs fit in memory. In case the two inputs are larger than memory, it is still possible to use the double-pipelined join technique as usual until available memory is full.

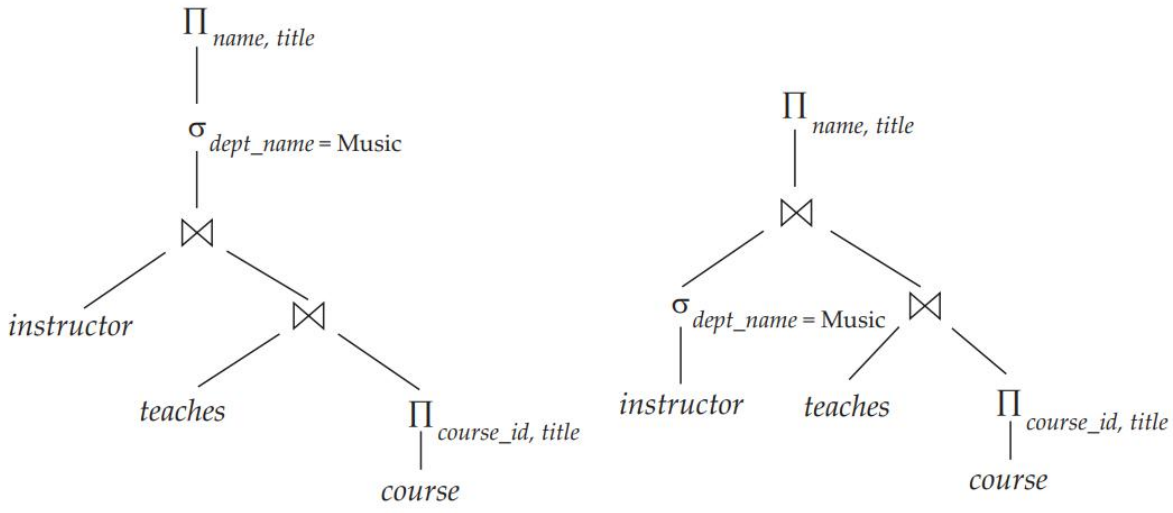
Query Optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

Overview

Consider the following relational-algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."

$$\Pi_{name,title}\left(\sigma_{dept\_name=\text{"Music"}}\left(instructor \bowtie \left(teaches \bowtie \Pi_{course\_id,title}(course)\right)\right)\right)$$



(a) Initial expression tree (b) Transformed expression tree

Figure 13.1 Equivalent expressions.

By reducing the number of tuples of the instructor relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression:

$$\Pi_{name,title}\left(\left(\sigma_{dept\_name=\text{"Music"}}\left(instructor\right)\right) \bowtie \left(teaches \bowtie \Pi_{course\_id,title}(course)\right)\right)$$

which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure 13.1 depicts the initial and transformed expressions.

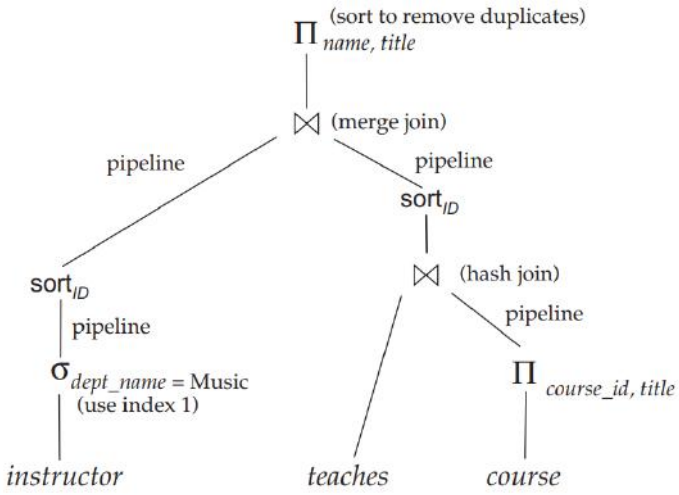


Figure 13.2 An evaluation plan.

Generation of query-evaluation plans involves three steps:

(1) generating expressions that are logically equivalent to the given expression.

(2) annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and

(3) estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least.

Transformation of Relational Expressions

Two relational-algebra expressions are said to be equivalent if, on every legal database instance, the two expressions generate the same set of tuples. (Recall that a legal database instance is one that satisfies all the integrity constraints specified in the database schema.) Note that the order of the tuples is irrelevant; the two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

Equivalence Rules:

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa—that is, we can replace an expression of the second form by an expression of the first form—since the two expressions generate the same result on any valid database. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

We now list a number of general equivalence rules on relational-algebra expressions. Some of the equivalences listed appear in Figure 13.3. We use θ, θ_1, θ_2, and so on to denote predicates, L_1, L_2, L_3, and so on to denote lists of attributes, and E, E_1, E_2, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can also be referred to as a cascade of .

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

      This expression is just the definition of the theta join.

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. Theta-join operations are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

6. a. Natural-join operations are associative.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

   b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where Θ2 involves attributes from only E2 and E3. Any of these conditions may be empty; hence, it follows that the Cartesian product (×) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:

a. It distributes when all the attributes in selection condition Θ0 involve only the attributes of one of the expressions (say, E1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

b. It distributes when selection condition Θ1 involves only the attributes of E1 and Θ2 involves only the attributes of E2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

Similarly, the preceding equivalence, with − replaced with either ∪ or ∩, also holds. Further:

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

The preceding equivalence, with − replaced by ∩, also holds, but does not hold if − is replaced by ∪.

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Examples of Transformations:

We now illustrate the use of the equivalence rules. We use our university example with the relation schemas:

instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

In our example, the expression:

$$\Pi_{name,title}\ (\sigma_{dept\_name = \text{“Music”}}\ (instructor\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course))))$$

was transformed into the following expression:

$$\Pi_{name,title}\ ((\sigma_{dept\_name = \text{“Music”}}\ (instructor))\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course)))$$

which is equivalent to our original algebra expression, but generates smaller intermediate relations.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to instructors who have taught a course in 2009. The new relational-algebra query is:

$$\Pi_{name,title}\ (\sigma_{dept\_name = \text{“Music”} \wedge year\ =\ 2009}\ (instructor\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course))))$$

We cannot apply the selection predicate directly to the *instructor* relation, since the predicate involves attributes of both the *instructor* and *teaches* relations. However, we can first apply rule 6.a (associativity of natural join) to transform the join $instructor\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course))$ into $(instructor\ \bowtie\ teaches)\ \bowtie\ \Pi_{course\_id,title}(course)$:

$$\Pi_{name,title}\ (\sigma_{dept\_name = \text{“Music”} \wedge year\ =\ 2009}\ ((instructor \bowtie teaches)\ \bowtie\ \Pi_{course\_id,title}(course)))$$

Then, using rule 7.a, we can rewrite our query as:

$$\Pi_{name,title}\ ((\sigma_{dept\_name = \text{“Music”} \wedge year\ =\ 2009}\ (instructor \bowtie teaches))\ \bowtie\ \Pi_{course\_id,title}(course))$$

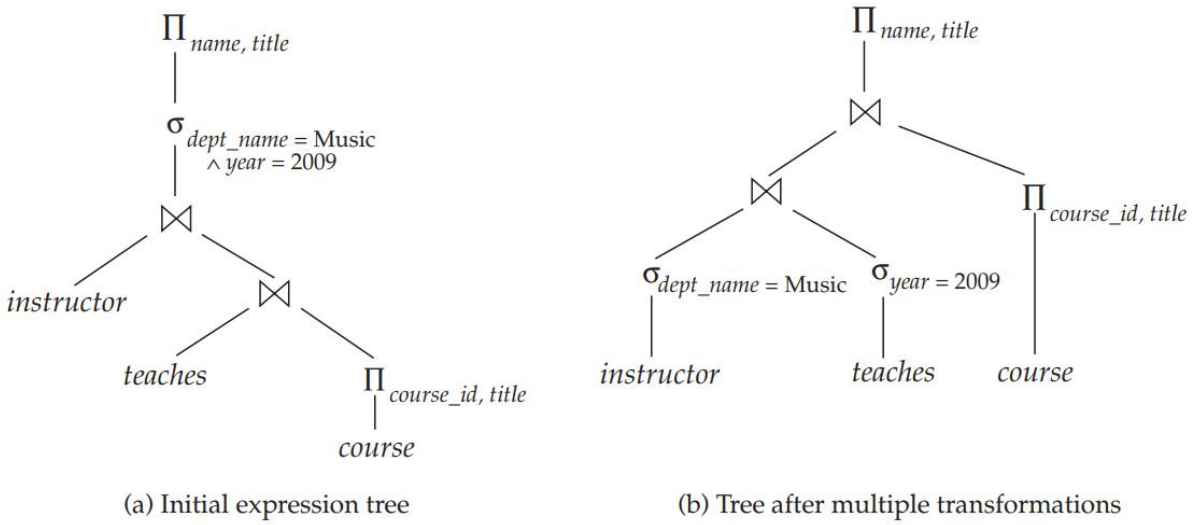
Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

$$\sigma_{dept\_name = \text{“Music”}}\ (\sigma_{year\ =\ 2009}\ (instructor \bowtie teaches))$$

Both of the preceding expressions select tuples with $dept\_name$ = “Music” and *course_id* = 2009. However, the latter form of the expression provides a new opportunity to apply Rule 7.a (“perform selections early”), resulting in the subexpression:

$$\sigma_{dept\_name = \text{“Music”}}\ (instructor)\ \bowtie\ \sigma_{year\ =\ 2009}\ (teaches)$$

Now consider the following form of our example query:



(a) Initial expression tree

(b) Tree after multiple transformations

Figure 13.4 Multiple transformations.

$$\Pi_{name, title}\ ((\sigma_{dept\_name = \text{“Music”}}\ (instructor) \bowtie teaches) \bowtie \Pi_{course\_id, title}(course))$$

When we compute the subexpression:

$$(\sigma_{dept\_name = \text{“Music”}}\ (instructor) \bowtie teaches)$$

we obtain a relation whose schema is:

$$(ID, name, dept\_name, salary, course\_id, sec\_id, semester, year)$$

we can modify the expression to:

$$\Pi_{name, title}\ ((\Pi_{name, course\_id}\ ((\sigma_{dept\_name = \text{“Music”}}\ (instructor)) \bowtie teaches) \bowtie \Pi_{course\_id, title}(course))$$

The projection $\Pi_{name, course\_id}$ reduces the size of the intermediate join results.

Join Ordering:

A good ordering of join operations is important for reducing the size of temporary results; hence, most query optimizers pay a lot of attention to the join order. In equivalence rule 6.a, the natural-join operation is associative. Thus, for all relations r1, r2, and r3:

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

Although these expressions are equivalent, the costs of computing them may differ. Consider again the expression:

$$\Pi_{name,title}\ ((\sigma_{dept\_name = \text{“Music”}}\ (instructor)) \bowtie teaches \bowtie \Pi_{course\_id,title}(course))$$

We could choose to compute $teaches \bowtie \Pi_{course\_id,title}(course)$ first, and then to join the result with:

$$\sigma_{dept\_name = \text{“Music”}}\ (instructor)$$

However, $teaches \bowtie \Pi_{course\_id,title}(course)$ is likely to be a large relation, since it contains one tuple for every course taught. In contrast:

$$\sigma_{dept\_name = \text{“Music”}}\ (instructor) \bowtie teaches$$

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations r_1 and r_2:

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

That is, natural join is commutative (equivalence rule 5).

Using the associativity and commutativity of the natural join (rules 5 and 6), consider the following relational-algebra expression:

$$(instructor \bowtie \Pi_{course\_id,title}(course)) \bowtie teaches$$

Note that there are no attributes in common between $\Pi_{course\_id,title}(course)$ and *instructor*, so the join is just a Cartesian product. If there are a tuples in *instructor* and b tuples in $\Pi_{course\_id,title}(course)$, this Cartesian product generates $a * b$ tuples, one for every possible pair of instructor tuple and course (without regard for whether the instructor taught the course). This Cartesian product would produce a very large temporary relation. However, if the user had entered the preceding expression, we could use the associativity and commutativity of the natural join to transform this expression to the more efficient expression:

$$(instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)$$

Enumeration of Equivalent Expressions:

Optimizers can greatly reduce both the space and time cost, using two key ideas.

1. If we generate an expression E’ from an expression E1 by using an equivalence rule on subexpression ei , then E’ and E1 have identical subexpressions.

```
procedure genAllEquivalent(E)
EQ = {E}
repeat
    Match each expression E_i in EQ with each equivalence rule R_j
    if any subexpression e_i of E_i matches one side of R_j
        Create a new expression E' which is identical to E_i, except that
            e_i is transformed to match the other side of R_j
        Add E' to EQ if it is not already present in EQ
until no new expression can be added to EQ
```

Figure 13.5 Procedure to generate all equivalent expressions.

except for ei and its transformation. Even ei and its transformed version usually share many identical subexpressions. Expression-representation techniques that allow both expressions to point to shared subexpressions can reduce the space requirement significantly. 2. It is not always necessary to generate every expression that can be generated with the equivalence rules. If an optimizer takes cost estimates of evaluation into account, it may be able to avoid examining some of the expressions.'

Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as a ⋈ (b ⋈ c) to estimate the cost of joining a with (b ⋈ c), we need to have estimates of statistics such as the size of b ⋈ c.

- **Catalog Information:**

The database-system catalog stores the following statistical information about database relations:

- n_r, the number of tuples in the relation r.
- b_r, the number of blocks containing tuples of relation r.
- l_r, the size of a tuple of relation r in bytes.
- f_r, the blocking factor of relation r—that is, the number of tuples of relation r that fit into one block.
- $V(A, r)$, the number of distinct values that appear in the relation r for attribute A. This value is the same as the size of $\Pi_A(r)$. If A is a key for relation r, $V(A, r)$ is n_r.

The last statistic, $V(A, r)$, can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, $\mathcal{A}$, $V(\mathcal{A}, r)$ is the size of $\Pi_{\mathcal{A}}(r)$.

If we assume that the tuples of relation r are stored together physically in a file, the following equation holds:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Statistics about indices, such as the heights of B^+-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

For instance, most databases store the distribution of values for each attribute as a histogram: in a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. Figure 13.6 shows an example of a histogram for an integer-valued attribute that takes values in the range 1 to 25.

As an example of a histogram, the range of values for an attribute age of a relation person could be divided into 0–9, 10–19, . . . , 90–99 (assuming a maximum age of 99). With each range we store a count of the number of person tuples whose age values lie in that range, and the number of distinct age values that lie in that range.

There are several types of histograms used in database systems. For example, an equi-width histogram divides the range of values into equal-sized ranges, whereas an equi-depth histogram adjusts the boundaries of the ranges such that each range has the same number of values.

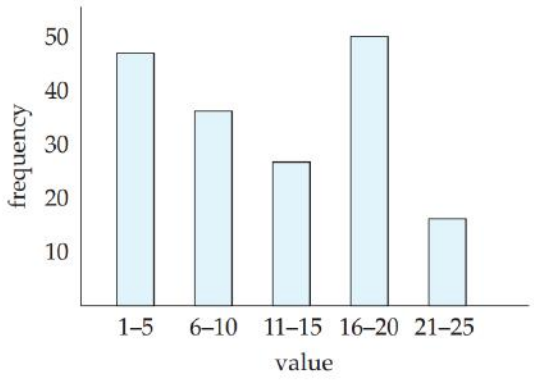


Figure 13.6 Example of histogram.

Selection Size Estimation:

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

• **$\sigma_{A=a}(r)$:** If we assume uniform distribution of values (that is, each value appears with equal probability), the selection result can be estimated to have nr /V(A,r) tuples, assuming that the value a appears in attribute A of some record of r. The assumption that the value a in the

selection appears in some record is generally true, and cost estimates often make it implicitly. However, it is often not realistic to assume that each value appears with equal probability. The course id attribute in the takes relation is an example where the assumption is not valid. It is reasonable to expect that a popular undergraduate course will have many more students than a smaller specialized graduate course. Therefore, certain course id values appear with greater probability than do others.

• **$\sigma_{A \le v}(r)$:** Consider a selection of the form $\sigma_{A \le v}(r)$. If the actual value used in the comparison (v) is available at the time of cost estimation, a more accurate estimate can be made. The lowest and highest values (min(A,r) and max(A,r)) for the attribute can be stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will satisfy the condition A ≤ v as 0 if v < min(A,r), as nr if v ≥ max(A,r), and:

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

• **Complex selections:**

◦ **Conjunction:** A conjunctive selection is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

We can estimate the result size of such a selection: For each θ_i, we estimate the size of the selection $\sigma_{\theta_i}(r)$, denoted by s_i, as described previously. Thus, the probability that a tuple in the relation satisfies selection condition θ_i is s_i / n_r.

we estimate the number of tuples in the full selection as:

$$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

◦ **Disjunction:** A disjunctive selection is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

As before, let s_i / n_r denote the probability that a tuple satisfies condition θ_i. The probability that the tuple will satisfy the disjunction is then 1 minus the probability that it will satisfy *none* of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplying this value by n_r gives us the estimated number of tuples that satisfy the selection.

- **Negation:** In the absence of nulls, the result of a selection $\sigma_{\neg\theta}(r)$ is simply the tuples of r that are not in $\sigma_\theta(r)$. We already know how to estimate the number of tuples in $\sigma_\theta(r)$. The number of tuples in $\sigma_{\neg\theta}(r)$ is therefore estimated to be $n(r)$ minus the estimated number of tuples in $\sigma_\theta(r)$.

Join Size Estimation:

In this section, we see how to estimate the size of the result of a join.

The Cartesian product $r \times s$ contains $n_r * n_s$ tuples. Each tuple of $r \times s$ occupies $l_r + l_s$ bytes, from which we can calculate the size of the Cartesian product.

Estimating the size of a natural join is somewhat more complicated than estimating the size of a selection or of a Cartesian product. Let $r(R)$ and $s(S)$ be relations.

- If $R \cap S = \emptyset$—that is, the relations have no attribute in common—then $r \bowtie s$ is the same as $r \times s$, and we can use our estimation technique for Cartesian products.
- If $R \cap S$ is a key for R, then we know that a tuple of s will join with at most one tuple from r. Therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s. The case where $R \cap S$ is a key for S is symmetric to the case just described. If $R \cap S$ forms a foreign key of S, referencing R, the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
- The most difficult case is when $R \cap S$ is a key for neither R nor S. In this case, we assume, as we did for selections, that each value appears with equal probability. Consider a tuple t of r, and assume $R \cap S = \{A\}$. We estimate that tuple t produces:

$$\frac{n_s}{V(A, s)}$$

tuples in $r \bowtie s$, since this number is the average number of tuples in s with a given value for the attributes A. Considering all the tuples in r, we estimate that there are:

$$\frac{n_r * n_s}{V(A, s)}$$

tuples in $r \bowtie s$. Observe that, if we reverse the roles of r and s in the preceding estimate, we obtain an estimate of:

$$\frac{n_r * n_s}{V(A, r)}$$

To illustrate all these ways of estimating join sizes, consider the expression:

$$student \bowtie takes$$

Assume the following catalog information about the two relations:

- $n_{student} = 5000$.
- $f_{student} = 50$, which implies that $b_{student} = 5000/50 = 100$.
- $n_{takes} = 10000$.
- $f_{takes} = 25$, which implies that $b_{takes} = 10000/25 = 400$.
- $V(ID, takes) = 2500$, which implies that only half the students have taken any course (this is unrealistic, but we use it to show that our size estimates are correct even in this case), and on average, each student who has taken a course has taken four courses.

Size Estimation for Other Operations:

We outline below how to estimate the sizes of the results of other relational-algebra operations.

- **Projection:** The estimated size (number of records or number of tuples) of a projection of the form $\Pi_A(r)$ is $V(A, r)$, since projection eliminates duplicates.
- **Aggregation:** The size of ${}_A\mathcal{G}_F(r)$ is simply $V(A, r)$, since there is one tuple in ${}_A\mathcal{G}_F(r)$ for each distinct value of A.
- **Set operations:** If the two inputs to a set operation are selections on the same relation, we can rewrite the set operation as disjunctions, conjunctions, or negations. For example, $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ can be rewritten as $\sigma_{\theta_1 \vee \theta_2}(r)$. Similarly, we can rewrite intersections as conjunctions, and we can rewrite set difference by using negation, so long as the two relations participating in the set operations are selections on the same relation. We can then use the estimates for selections involving conjunctions, disjunctions, and negation in Section 13.3.2.

  If the inputs are not selections on the same relation, we estimate the sizes this way: The estimated size of $r \cup s$ is the sum of the sizes of r and s. The estimated size of $r \cap s$ is the minimum of the sizes of r and s. The estimated size of $r - s$ is the same size as r. All three estimates may be inaccurate, but provide upper bounds on the sizes.
- **Outer join:** The estimated size of $r \mathbin{⟕} s$ is the size of $r \bowtie s$ plus the size of r; that of $r \mathbin{⟖} s$ is symmetric, while that of $r \mathbin{⟗} s$ is the size of $r \bowtie s$ plus the sizes of r and s. All three estimates may be inaccurate, but provide upper bounds on the sizes.

Estimation of Number of Distinct Values:

For selections, the number of distinct values of an attribute (or set of attributes) A in the result of a selection, $V(A, \sigma_\theta(r))$, can be estimated in these ways:

- If the selection condition θ forces A to take on a specified value (e.g., $A = 3$), $V(A, \sigma_\theta(r)) = 1$.
- If θ forces A to take on one of a specified set of values (e.g., $(A = 1 \vee A = 3 \vee A = 4)$), then $V(A, \sigma_\theta(r))$ is set to the number of specified values.
- If the selection condition θ is of the form $A\ op\ v$, where op is a comparison operator, $V(A, \sigma_\theta(r))$ is estimated to be $V(A, r) * s$, where s is the selectivity of the selection.
- In all other cases of selections, we assume that the distribution of A values is independent of the distribution of the values on which selection conditions are specified, and we use an approximate estimate of $\min(V(A, r), n_{\sigma_\theta(r)})$. A more accurate estimate can be derived for this case using probability theory, but the above approximation works fairly well.

For joins, the number of distinct values of an attribute (or set of attributes) A in the result of a join, $V(A, r \bowtie s)$, can be estimated in these ways:

- If all attributes in A are from r, $V(A, r \bowtie s)$ is estimated as $\min(V(A, r), n_{r \bowtie s})$, and similarly if all attributes in A are from s, $V(A, r \bowtie s)$ is estimated to be $\min(V(A, s), n_{r \bowtie s})$.
- If A contains attributes $A1$ from r and $A2$ from s, then $V(A, r \bowtie s)$ is estimated as:

$$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

Choice of Evaluation Plans

Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms. An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated.

A cost-based optimizer explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.

Cost-Based Join Order Selection:

For a complex join query, the number of different query plans that are equivalent to the query can be large. As an illustration, consider the expression:

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

where the joins are expressed without any ordering. With $n = 3$, there are 12 different join orderings:

$r_1 \bowtie (r_2 \bowtie r_3)$	$r_1 \bowtie (r_3 \bowtie r_2)$	$(r_2 \bowtie r_3) \bowtie r_1$	$(r_3 \bowtie r_2) \bowtie r_1$
$r_2 \bowtie (r_1 \bowtie r_3)$	$r_2 \bowtie (r_3 \bowtie r_1)$	$(r_1 \bowtie r_3) \bowtie r_2$	$(r_3 \bowtie r_1) \bowtie r_2$
$r_3 \bowtie (r_1 \bowtie r_2)$	$r_3 \bowtie (r_2 \bowtie r_1)$	$(r_1 \bowtie r_2) \bowtie r_3$	$(r_2 \bowtie r_1) \bowtie r_3$

In general, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. (We leave the computation of this expression for you to do in Exercise 13.10.) For joins involving small numbers of relations, this number is acceptable; for example, with $n = 5$, the number is 1680. However, as n increases, this number rises quickly. With $n = 7$, the number is 665,280; with $n = 10$, the number is greater than 17.6 billion!

Cost-Based Optimization with Equivalence Rules:

The procedure for generating equivalent expressions can be modified to generate all possible evaluation plans as follows: A new class of equivalence rules, called physical equivalence rules, is added that allows a logical operation, such as a join, to be transformed to a physical operation, such as a hash join, or a nested-loops join. By adding such rules to the original set of equivalence rules, the procedure can generate all possible evaluation plans. The cost estimation techniques we have seen earlier can then be used to choose the optimal (that is, the least-cost) plan.

To make the approach work efficiently requires the following:

1. A space-efficient representation of expressions that avoids making multiple copies of the same subexpressions when equivalence rules are applied.
2. Efficient techniques for detecting duplicate derivations of the same expression.
3. A form of dynamic programming based on **memoization**, which stores the optimal query evaluation plan for a subexpression when it is optimized for the first time; subsequent requests to optimize the same subexpression are handled by returning the already memoized plan.
4. Techniques that avoid generating all possible equivalent plans, by keeping track of the cheapest plan generated for any subexpression up to any point of time, and pruning away any plan that is more expensive than the cheapest plan found so far for that subexpression.

Heuristics in Optimization:

An example of a heuristic rule is the following rule for transforming relational algebra queries:

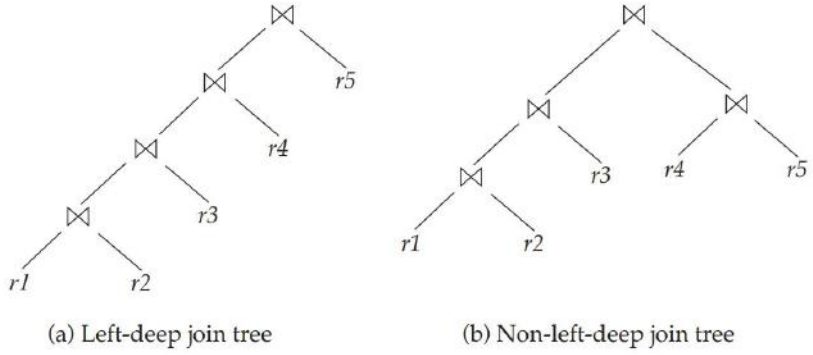
• Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation.

• Perform projections early.

It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples. An example similar to the one used for the selection heuristic should convince you that this heuristic does not always reduce the cost.

Most practical query optimizers have further heuristics to reduce the cost of optimization. For example, many query optimizers, such as the System R optimizer,3 do not consider all join orders, but rather restrict the search to particular kinds of join orders. The System R optimizer considers only those join orders where the right operand of each join is one of the initial relations r1,...,rn. Such join orders are called left-deep join orders. Left-deep join orders are particularly convenient for pipelined evaluation, since the right operand is a stored relation, and thus only one input to each join is pipelined.



(a) Left-deep join tree (b) Non-left-deep join tree

Figure 13.8 Left-deep join trees.

Optimizing Nested Subqueries**:

The parameters are the variables from an outer level query that are used in the nested subquery (these variables are called correlation variables). For instance, suppose we have the following query, to find the names of all instructors who taught a course in 2007:

```
select name
from instructor
where exists (select *
              from teaches
              where instructor.ID = teaches.ID
                    and teaches.year = 2007);
```

Conceptually, the subquery can be viewed as a function that takes a parameter (here, *instructor.ID*) and returns the set of all courses taught in 2007 by instructors (with the same *ID*).

This technique for evaluating a query with a nested subquery is called correlated evaluation. Correlated evaluation is not very efficient, since the subquery is separately evaluated for each tuple in the outer level query. A large number of random disk I/O operations may result.

As an example of transforming a nested subquery into a join, the query in the preceding example can be rewritten as:

```
select name
from instructor, teaches
where instructor.ID = teaches.ID and teaches.year = 2007;
```

In the example, the nested subquery was very simple. In general, it may not be possible to directly move the nested subquery relations into the from clause of the outer query. Instead, we create a temporary relation that contains the results of the nested query without the selections using correlation variables from the outer query, and join the temporary table with the outer level query. For instance, a query of the form:

```
select ...
from L1
where P1 and exists (select *
                     from L2
                     where P2);
```

where P_2 is a conjunction of simpler predicates, can be rewritten as:

```
create table t1 as
        select distinct V
                from L2
                where P2^1;
select ...
from L1, t1
where P1 and P2^2;
```

In our example, the original query would have been transformed to:

```
create table t1 as
        select distinct ID
        from teaches
        where year = 2007;
select name
from instructor, t1
where t1.ID = instructor.ID;
```

Materialized Views**

When a view is defined, normally the database stores only the query defining the view. In contrast, a materialized view is a view whose contents are computed and stored. Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents.

Materialized views are important for improving performance in some applications. Consider this view, which gives the total salary in each department:

```
create view department_total_salary(dept_name, total_salary) as
select dept_name, sum (salary)
from instructor
group by dept_name;
```

View Maintenance:

The task of keeping a materialized view up-to-date with the underlying data is known as view maintenance.

A better option is to modify only the affected parts of the materialized view, which is known as **incremental view maintenance**.

Most database systems perform **immediate view maintenance**; that is, incremental view maintenance is performed as soon as an update occurs, as part of the updating transaction.

Some database systems also support **deferred view maintenance**, where view maintenance is deferred to a later time; for example, updates may be collected throughout a day, and materialized views may be updated at night. This approach reduces the overhead on update transactions.

Incremental View Maintenance:

The changes to a relation that can cause a materialized view to become outof-date are inserts, deletes, and updates.

- **Join Operation:**

Consider the materialized view $v = r \bowtie s$. Suppose we modify r by inserting a set of tuples denoted by i_r. If the old value of r is denoted by r^{old}, and the new value of r by r^{new}, $r^{new} = r^{old} \cup i_r$. Now, the old value of the view, v^{old}, is given by $r^{old} \bowtie s$, and the new value v^{new} is given by $r^{new} \bowtie s$. We can rewrite $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$, which we can again rewrite as $(r^{old} \bowtie s) \cup (i_r \bowtie s)$. In other words:

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

Thus, to update the materialized view v, we simply need to add the tuples $i_r \bowtie s$ to the old contents of the materialized view. Inserts to s are handled in an exactly symmetric fashion.

Now suppose r is modified by deleting a set of tuples denoted by d_r. Using the same reasoning as above, we get:

$$v^{new} = v^{old} - (d_r \bowtie s)$$

Deletes on s are handled in an exactly symmetric fashion.

- **Selection and Projection Operations:**

Consider a view $v = \sigma_\theta(r)$. If we modify r by inserting a set of tuples i_r, the new value of v can be computed as:

$$v^{new} = v^{old} \cup \sigma_\theta(i_r)$$

Similarly, if r is modified by deleting a set of tuples d_r, the new value of v can be computed as:

$$v^{new} = v^{old} - \sigma_\theta(d_r)$$

Projection is a more difficult operation with which to deal. Consider a materialized view $v = \Pi_A(r)$. Suppose the relation r is on the schema $R = (A, B)$, and r contains two tuples $(a, 2)$ and $(a, 3)$. Then, $\Pi_A(r)$ has a single tuple (a). If we delete the tuple $(a, 2)$ from r, we cannot delete the tuple (a) from $\Pi_A(r)$: If we did so, the result would be an empty relation, whereas in reality $\Pi_A(r)$ still has a single tuple (a). The reason is that the same tuple (a) is derived in two ways, and deleting one tuple from r removes only one of the ways of deriving (a); the other is still present.

- **Aggregation Operations:**

Aggregation operations proceed somewhat like projections. The aggregate operations in SQL are count, sum, avg, min, and max:

- **count**: Consider a materialized view $v = {}_A\mathcal{G}_{count(B)}(r)$, which computes the count of the attribute B, after grouping r by attribute A.
- **sum**: Consider a materialized view $v = {}_A\mathcal{G}_{sum(B)}(r)$.
- **avg**: Consider a materialized view $v = {}_A\mathcal{G}_{avg(B)}(r)$.
- **min, max**: Consider a materialized view $v = {}_A\mathcal{G}_{min(B)}(r)$. (The case of **max** is exactly equivalent.)

Other Operations:

The set operation intersection is maintained as follows: Given materialized view v = r ∩ s, when a tuple is inserted in r we check if it is present in s, and if so we add it to v. If a tuple is deleted from r, we delete it from the intersection if it is present. The other set operations, union and set difference, are handled in a similar fashion;

Handling Expressions:

So far we have seen how to update incrementally the result of a single operation. To handle an entire expression, we can derive expressions for computing the incremental change to the result of each subexpression, starting from the smallest subexpressions.

For example, suppose we wish to incrementally update a materialized view $E_1 \bowtie E_2$ when a set of tuples i_r is inserted into relation r. Let us assume r is used in E_1 alone. Suppose the set of tuples to be inserted into E_1 is given by expression D_1. Then the expression $D_1 \bowtie E_2$ gives the set of tuples to be inserted into $E_1 \bowtie E_2$.

- **Query Optimization and Materialized Views:**

Query optimization can be performed by treating materialized views just like regular relations. However, materialized views offer further opportunities for optimization:

• Rewriting queries to use materialized views:

• Replacing a use of a materialized view with the view definition:

- **Materialized View and Index Selection:**

Another related optimization problem is that of materialized view selection, namely, "What is the best set of views to materialize?" This decision must be made on the basis of the system workload, which is a sequence of queries and updates that reflects the typical load on the system. One simple criterion would be to select a set of materialized views that minimizes the overall execution time of the workload of queries and updates, including the time taken to maintain the materialized views.

Indices are just like materialized views, in that they too are derived data, can speed up queries, and may slow down updates. Thus, the problem of index selection is closely related to that of materialized view selection, although it is simpler.

Advanced Topics in Query Optimization**

- **Top-K Optimization:**

Many queries fetch results sorted on some attributes, and require only the top K results for some K. Sometimes the bound K is specified explicitly. For example, some databases support a limit K clause which results in only the top K results being returned by the query.

- **Join Minimization:**

When queries are generated through views, sometimes more relations are joined than are needed for computation of the query. For example, a view v may include the join of instructor and department, but a use of the view v may use only attributes from instructor. The join attribute dept name of instructor is a foreign key referencing department. Assuming that instructor.dept name has been declared not null, the join with department can be dropped, with no impact on the query. For, under the above assumption, the join with department does not eliminate any tuples from instructor, nor does it result in extra copies of any instructor tuple.

Dropping a relation from a join as above is an example of join minimization. In fact, join minimization can be performed in other situations as well. See the bibliographical notes for references on join minimization.

Optimization of Updates:

Update queries often involve subqueries in the set and where clauses, which must also be taken into account in optimizing the update. Updates that involve a selection on the updated column (e.g., give a 10 percent salary raise to all employees whose salary is ≥ $100,000) must be handled carefully. If the update is done while the selection is being evaluated by an index scan, an updated tuple may be reinserted in the index ahead of the scan and seen again by the

scan; the same employee tuple may then get incorrectly updated multiple times (an infinite number of times, in this case).

The problem of an update affecting the execution of a query associated with the update is known as the Halloween problem (named so because it was first recognized on a Halloween day, at IBM).

- **Multiquery Optimization and Shared Scans:**

Complex queries may in fact have subexpressions repeated in different parts of the query, which can be similarly exploited, to reduce query evaluation cost. Such optimization is known as multiquery optimization.

Common subexpression elimination optimizes subexpressions shared by different expressions in a program, by computing and storing the result, and reusing it wherever the subexpression occurs. Common subexpression elimination is a standard optimization applied on arithmetic expressions by programming language compilers.

The shared-scan optimization works as follows: Instead of reading the relation repeatedly from disk, once for each query that needs to scan a relation, data are read once from disk, and pipelined to each of the queries. The shared-scan optimization is particularly useful when multiple queries perform a scan on a single large relation (typically a “fact table”).

- **Parametric Query Optimization:**

In parametric query optimization, a query is optimized without being provided specific values for its parameters, for example, dept name in the preceding example. The optimizer then outputs several plans, each optimal for a different parameter value. A plan would be output by the optimizer only if it is optimal for some possible value of the parameters. The set of alternative plans output by the optimizer are stored. When a query is submitted with specific values for its parameters, instead of performing a full optimization, the cheapest plan from the set of alternative plans computed earlier is used. Finding the cheapest such plan usually takes much less time than reoptimization.

UNIT-5

Transaction Management, Concurrency control and Recovery System

Transaction Management: Transactions: Concept, A Simple Transactional Model, Storage Structures, Transaction Atomicity and Durability, Transaction Isolation, Serializability, Isolation and Atomicity, Transaction Isolation Levels, Implementation of Isolation Levels, Transactions as SQL Statements.

Concurrency Control: Lock-based Protocols, Deadlock Handling, Multiple granularity, Timestamp-based Protocols, and Validation-based Protocols.

Recovery System: Failure Classification, Storage, Recovery and Atomicity, Recovery Algorithm, Buffer Management, Failure with Loss of Nonvolatile Storage, Early Lock Release and Logical Undo Operations.

Transactions

Collections of operations that form a single logical unit of work are called transactions. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency.

Transaction Concept

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC. A transaction is delimited by statements (or function calls) of the form begin transaction and end transaction. The transaction consists of all operations executed between the begin transaction and end transaction.

The database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.

- **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

- **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

• **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID** properties; the acronym is derived from the first letter of each of the four properties.

A Simple Transaction Model

We shall illustrate the transaction concept using a simple bank application consisting of several accounts and a set of transactions that access and update those accounts. Transactions access data using two operations:

• read(X), which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.

• write(X), which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

$$
\begin{aligned}
T_i:\ & \text{read}(A);\\
& A := A - 50;\\
& \text{write}(A);\\
& \text{read}(B);\\
& B := B + 50;\\
& \text{write}(B).
\end{aligned}
$$

Let us now consider each of the ACID properties.

• **Consistency:** The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

• **Atomicity:** Suppose that, just before the execution of transaction Ti , the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti , a failure occurs that prevents Ti from completing its execution successfully. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum A + B is no longer preserved.

• **Durability:** Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be

the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We can guarantee durability by ensuring that either:

1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

• **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

Storage Structure

• **Volatile storage.** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

• **Nonvolatile storage.** Information residing in nonvolatile storage survives system crashes. Examples of nonvolatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage. At the current state of technology, nonvolatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failure which may result in loss of information.

• **Stable storage.** Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically never cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with

independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

Transaction Atomicity and Durability

A transaction may not always complete its execution successfully. Such a transaction is termed **aborted.**

Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.

We need to be more precise about what we mean by successful completion of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

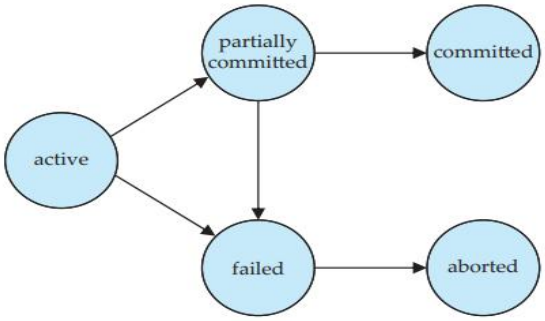


Figure 14.1 State diagram of a transaction.

The state diagram corresponding to a transaction appears in Figure 14.1. We say that a transaction has committed only if it has entered the committed state.

Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

• It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

• It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

Transaction Isolation

Transaction-processing systems usually allow multiple transactions to run concurrently. it is far easier to insist that transactions run serially— that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

• **Improved throughput and resource utilization.** A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system— that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• **Reduced waiting time.** There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system. The database system must

control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes.

Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as:

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as:

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts A and B are $1000 and $2000, respectively.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 14.2 Schedule 1 — a serial schedule in which T_1 is followed by T_2.

Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears in Figure 14.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure 14.2 takes place, are $855 and $2145, respectively. Thus, the

total amount of money in accounts A and B— that is, the sum A + B—is preserved after the execution of both transactions.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Figure 14.3 Schedule 2 — a serial schedule in which T_2 is followed by T_1.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure 14.3. Again, as expected, the sum A + B is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

The execution sequences just described are called schedules. They represent the chronological order in which instructions are executed in the system. These schedules are serial: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinatorics, we note that, for a set of n transactions, there exist n factorial (n!) different valid serial schedules.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 14.4 Schedule 3 — a concurrent schedule equivalent to schedule 1.

The two transactions are executed concurrently. One possible schedule appears in Figure 14.4. After this execution takes place, we arrive at the same state as the one in which the

transactions are executed serially in the order T1 followed by T2. The sum A + B is indeed preserved.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Figure 14.5 Schedule 4 — a concurrent schedule resulting in an inconsistent state.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 14.5. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are $950 and $2100, respectively. This final state is an inconsistent state, since we have gained $50 in the process of the concurrent execution. Indeed, the sum A + B is not preserved by the execution of the two transactions.

Serializability

In this section, we discuss different forms of schedule equivalence, but focus on a particular form called conflict serializability.

Let us consider a schedule S in which there are two consecutive instructions, I and J , of transactions Ti and Tj , respectively (i = j). If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule. However, if I and J refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:8

1. I = read(Q), J = read(Q). The order of I and J does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

2. I = read(Q), J = write(Q). If I comes before J , then Ti does not read the value of Q that is written by Tj in instruction J . If J comes before I, then Ti reads the value of Q that is written by Tj . Thus, the order of I and J matters.

3. I = write(Q), J = read(Q). The order of I and J matters for reasons similar to those of the previous case.

4. I = write(Q), J = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

We say that I and J conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3in Figure 14.6. The write(A) instruction of T1 conflicts with the read(A) instruction of T2. However, the write(A) instruction of T2 does not conflict with the read(B) instruction of T1, because the two instructions access different data items.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 14.6 Schedule 3 — showing only the read and write instructions

ith the read(B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 14.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure 14.7 Schedule 5 — schedule 3 after swapping of a pair of instructions.

Since the write(A) instruction of T2 in schedule 3 of Figure 14.6 does not conflict with the read(B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 14.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state. We continue to swap nonconflicting instructions:

• Swap the read(B) instruction of T1 with the read(A) instruction of T2.

• Swap the write(B) instruction of T1 with the write(A) instruction of T2.

• Swap the write(B) instruction of T1 with the read(A) instruction of T2.

The final result of these swaps, schedule 6 of Figure 14.8, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are conflict equivalent.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figure 14.9 Schedule 7.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1. Finally, consider schedule 7 of Figure 14.9; it consists of only the significant operations (that is, the read and write) of transactions T3 and T4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <T3,T4> or the serial schedule <T4,T3>.

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule S. We construct a directed graph, called a **precedence graph**, from S. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges Ti → Tj for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q).

2. Ti executes read(Q) before Tj executes write(Q).

3. Ti executes write(Q) before Tj executes write(Q).

If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S' equivalent to S, Ti must appear before Tj .

Figure 14.10 Precedence graph for (a) schedule 1 and (b) schedule 2.

For example, the precedence graph for schedule 1 in Figure 14.10a contains the single edge T1 → T2, since all the instructions of T1 are executed before the first instruction of T2 is executed. Similarly, Figure 14.10b shows the precedence graph for schedule 2 with the single edge T2 → T1, since all the instructions of T2 are executed before the first instruction of T1 is executed.

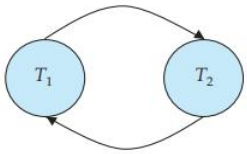


Figure 14.11 Precedence graph for schedule 4.

The precedence graph for schedule 4 appears in Figure 14.11. It contains the edge T1 → T2, because T1 executes read(A) before T2 executes write(A). It also contains the edge T2 → T1, because T2 executes read(B) before T1 executes write(B). If the precedence graph for S has a cycle, then schedule S is not conflict serializable.

If the graph contains no cycles, then the schedule S is conflict serializable.

A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called topological sorting.

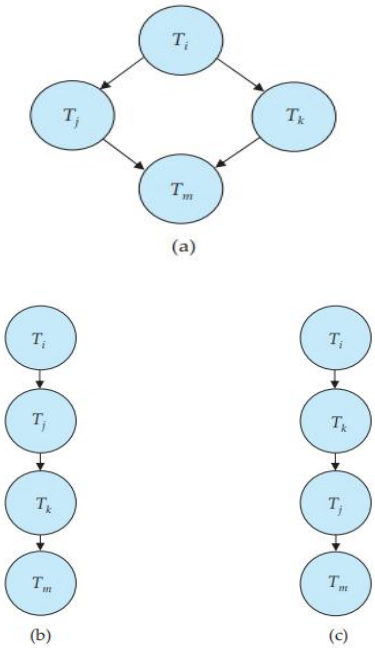


Figure 14.12 Illustration of topological sorting.

There are, in general, several possible linear orders that can be obtained through a topological sort. For example, the graph of Figure 14.12a has the two acceptable linear orderings shown in Figures 14.12b and 14.12c.

Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2 operations, where n is the number of vertices in the graph (that is, the number of transactions).

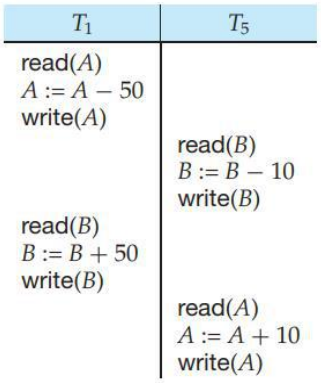


Figure 14.13 Schedule 8.

It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction T5, which transfers $10 from account B to account A. Let schedule 8 be as defined in Figure 14.13. We claim that schedule 8 is not conflict equivalent to the serial schedule <T1,T5> , since, in schedule 8, the write(B) instruction of T5 conflicts with the read(B) instruction of T1. This creates an edge T5 → T1 in the precedence graph. Similarly, we see that the write(A) instruction of T1 conflicts with the read instruction of T5 creating an edge T1 → T5. This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule <T1,T5> are the same—$960 and $2040, respectively.

Transaction Isolation and Atomicity

If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, the atomicity property requires that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted. To achieve this, we need to place restrictions on the type of schedules permitted in the system.

- **Recoverable Schedules:**

Consider the partial schedule 9 in Figure 14.14, in which T7 is a transaction that performs only one instruction: read(A). We call this a partial schedule because we have not included a commit or abort operation for T6. Notice that T7 commits immediately after executing the read(A) instruction. Thus, T7 commits while T6 is still in the active state. Now suppose that T6 fails before it commits. T7 has read the value of data item A written by T6. Therefore, we say that T7 is dependent on T6. Because of this, we must abort T7 to ensure atomicity. However, T7 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T6.

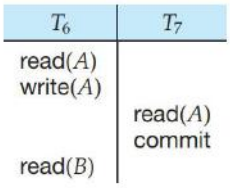


Figure 14.14 Schedule 9, a nonrecoverable schedule.

Schedule 9 is an example of a nonrecoverable schedule. A recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the commit operation of Tj . For the example of schedule 9 to be recoverable, T7 would have to delay committing until after T6 commits.

- **Cascadeless Schedules:**

Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti , we may have to roll back several transactions. Such situations occur if transactions have read data written by Ti . As an illustration, consider the partial schedule of Figure 14.15. Transaction T8 writes a value of A that is read by transaction T9. Transaction T9 writes a value of A that is read by transaction T10. Suppose that, at this point, T8 fails. T8 must be rolled back. Since T9 is dependent on T8, T9 must be rolled back. Since T10 is dependent on T9, T10 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascadeless schedules. Formally, a cascadeless schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the read operation of Tj . It is easy to verify that every cascadeless schedule is also recoverable.

Transaction Isolation Levels

The isolation levels specified by the SQL standard are as follows:

• **Serializable** usually ensures serializable execution. However, as we shall explain shortly, some database systems implement this isolation level in a manner that may, in certain cases, allow nonserializable executions.

• **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.

• **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.

• **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

All the isolation levels above additionally disallow dirty writes, that is, they disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Implementation of Isolation Levels

There are various concurrency-control policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions.

As a trivial example of a concurrency-control policy, consider this: A transaction acquires a lock on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are recoverable and cascadeless as well.

- **Locking:**

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes. Many transactions can hold shared locks on the same data item at the same time, but a transaction is allowed an exclusive lock on a data item only if no other transaction holds any lock (regardless of whether shared or exclusive) on the data item. This use of two modes of locks along with two-phase locking allows concurrent reading of data while still ensuring serializability.

- **Timestamps:**

Another category of techniques for the implementation of isolation assigns each transaction a timestamp, typically when it begins. For each data item, the system keeps two timestamps. The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item. The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item. Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses conflict. When this is not possible, offending transactions are aborted and restarted with a new timestamp.

- **Multiple Versions and Snapshot Isolation:**

There are a variety of multiversion concurrency control techniques. One in particular, called snapshot isolation, is widely used in practice.

In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins. It reads data from this private version and is thus isolated from the updates made by other transactions. If the transaction updates the database, that update appears only in its own version, not in the actual database itself. Information about these updates is saved so that the updates can be applied to the "real" database if the transaction commits.

Snapshot isolation ensures that attempts to read data never need to wait (unlike locking). Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting. Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking). Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

Transactions as SQL Statements

In our simple model, we assumed a set of data items exists. While our simple model allowed data-item values to be changed, it did not allow data items to be created or deleted. In SQL, however, insert statements create new data and delete statements delete data. These two statements are, in effect, write operations, since they change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model. As an example, consider the following SQL query on our university database that finds all instructors who earn more than $90,000.

SQL> select ID, name from instructor where salary > 90000;

Now assume that around the same time we are running our query, another user inserts a new instructor named "James" whose salary is $100,000.

SQL> insert into instructor values ('11111', 'James', 'Marketing', 100000);

In a concurrent execution of these transactions, it is intuitively clear that they conflict, but this is a conflict not captured by our simple model. This situation is referred to as the phantom phenomenon, because a conflict may exist on "phantom" data.

Let us consider again the query:

SQL> select ID, name from instructor where salary> 90000;

and the following SQL update:

SQL> update instructor set salary = salary * 0.9 where name = 'Wu';

In our example query above, the predicate is "salary > 90000", and an update of Wu's salary from $90,000 to a value greater than $90,000, or an update of Einstein's salary from a value greater that $90,000 to a value less than or equal to $90,000, would conflict with this predicate. Locking based on this idea is called predicate locking.

Concurrency Control

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called concurrency control schemes.

Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

- **Locks**

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. Shared. If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.

2. Exclusive. If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

	S	X
S	true	false
X	false	false

Figure 15.1 Lock-compatibility matrix comp.

To state this more generally, given a set of lock modes, we can define a compatibility function on them as follows: Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj (Ti ≠ Tj) currently holds a lock of mode B. If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 15.1. An element comp(A, B) of the matrix has the value true if and only if mode A is compatible with mode B.16

A transaction requests a shared lock on data item Q by executing the lockS(Q) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

To access a data item, transaction Ti must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrencycontrol manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, Ti is made to wait until all incompatible locks held by other transactions have been released.

Transaction Ti may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A (Figure 15.2). Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B (Figure 15.3).

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 15.2 Transaction T_1.

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 15.3 Transaction T_2.

Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 15.4, is possible. In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure 15.4 Schedule 1.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T3 corresponds to T1 with unlocking delayed (Figure 15.5). Transaction T4 corresponds to T2 with unlocking delayed (Figure 15.6).

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Figure 15.5 Transaction T_3 (transaction T_1 with unlocking delayed). **Figure 15.6** Transaction T_4 (transaction T_2 with unlocking delayed).

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 15.7 for T3 and T4. Since T3 is holding an exclusive mode lock on B and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B. Similarly, since T4 is holding a shared mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 15.7 Schedule 2.

- **Granting of Locks:**

Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock. Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish. But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

1. There is no other transaction holding a lock on Q in a mode that conflicts with M.
2. There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti. Thus, a lock request will never get blocked by a lock request that is made later.

- **The Two-Phase Locking Protocol:**

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.

2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions T3 and T4 are two phase. On the other hand, transactions T1 and T2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T3, we could move the unlock(B) instruction to just after the lock-X(A) instruction, and still retain the two-phase locking property.

Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 15.8. Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.

Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 15.8 Partial schedule under two-phase locking.

- **Implementation of Locking:**

A lock manager can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the lock table. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

Figure 15.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the

rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

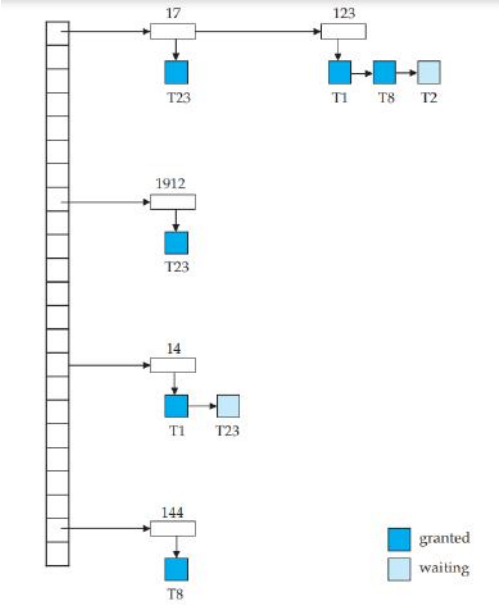


Figure 15.10 Lock table.

The lock manager processes requests this way:

• When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise, the request has to wait.

• When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

• If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction it releases all locks held by the aborted transaction.

- **Graph-Based Protocols:**

To acquire such prior knowledge, we impose a partial ordering → on the set D = {d1, d2,..., dh} of all data items. If di → dj , then any transaction accessing both di and dj must access di before accessing dj .

The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph.

In the tree protocol, the only lock instruction allowed is lock-X. Each transaction Ti can lock a data item at most once, and must observe the following rules:

1. The first lock by Ti may be on any data item.

2. Subsequently, a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti .

3. Data items may be unlocked at any time.

4. A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti .

To illustrate this protocol, consider the database graph of Figure 15.11. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

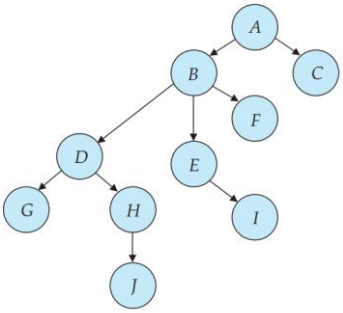
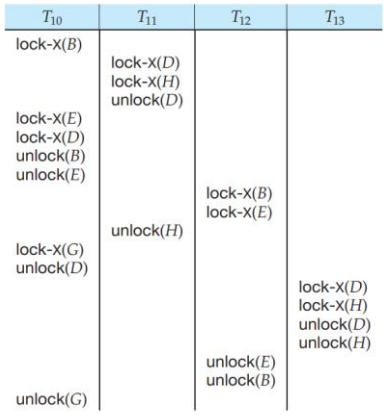


Figure 15.11 Tree-structured database graph.

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in Figure 15.12. Note that, during its execution, transaction T10 holds locks on two disjoint subtrees.

Observe that the schedule of Figure 15.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.



T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Figure 15.12 Serializable schedule under the tree protocol.

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions {T0, T1,..., Tn} such that T0 is waiting for a data item that T1 holds, and T1 is waiting for a data item that T2 holds, and ... , and Tn−1 is waiting for a data item that Tn holds, and Tn is waiting for a data item that T0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme.

- **Deadlock Prevention:**

Two different deadlock-prevention schemes using timestamps have been proposed:

1. The wait–die scheme is a nonpreemptive technique. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp smaller than that of Tj (that is, Ti is older than Tj). Otherwise, Ti is rolled back (dies).

For example, suppose that transactions T14, T15, and T16 have timestamps 5, 10, and 15, respectively. If T14 requests a data item held by T15, then T14 will wait. If T24 requests a data item held by T15, then T16 will be rolled back.

2. The wound–wait scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).

Returning to our example, with transactions T14, T15, and T16, if T14 requests a data item held by T15, then the data item will be preempted from T15, and T15 will be rolled back. If T16 requests a data item held by T15, then T16 will wait.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Another simple approach to deadlock prevention is based on lock timeouts. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.

Deadlock Detection and Recovery:

- ### Deadlock Detection:

Deadlocks can be described precisely in terms of a directed graph called a waitfor graph. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair Ti → Tj . If Ti → Tj is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.

When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .

To illustrate these concepts, consider the wait-for graph in Figure 15.13, which depicts the following situation:

• Transaction T17 is waiting for transactions T18 and T19.

• Transaction T19 is waiting for transaction T18.

• Transaction T18 is waiting for transaction T20.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction T20 is requesting an item held by T19. The edge T20 → T19 is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle: T18 → T20 → T19 → T18

implying that transactions T18, T19, and T20 are all deadlocked.

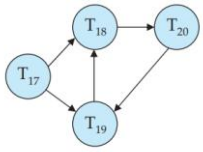


Figure 15.14 Wait-for graph with a cycle.

Recovery from Deadlock:

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. Selection of a victim. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Many factors may determine the cost of a rollback, including:

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.

b. How many data items the transaction has used.

c. How many more data items the transaction needs for it to complete. d. How many transactions will be involved in the rollback.

2. Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

The simplest solution is a total rollback: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such partial rollback requires the system to maintain additional information about the state of all the running transactions.

3. Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Multiple Granularity

What is needed is a mechanism to allow the system to define multiple levels of granularity. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

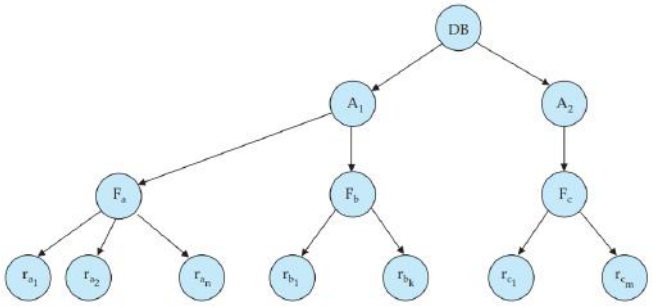


Figure 15.15 Granularity hierarchy.

As an illustration, consider the tree of Figure 15.15, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the twophase locking protocol, we shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction Ti gets an explicit lock on file Fc of Figure 15.15, in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of Fc explicitly.

- **intention lock modes:**

If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, Q—must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an **intention mode** associated with **shared mode,** and there is one with **exclusive mode.**

intention-shared (IS) mode: If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.

intention-exclusive (IX) mode: Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.

shared and intention-exclusive (SIX) mode: Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

The compatibility function for these lock modes is in Figure 15.16.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 15.16 Compatibility matrix.

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction Ti that attempts to lock a node Q must follow these rules:

1. Transaction Ti must observe the lock-compatibility function of Figure 15.16.

2. Transaction Ti must lock the root of the tree first, and can lock it in any mode.

3. Transaction Ti can lock a node Q in S or IS mode only if Ti currently has the parent of Q locked in either IX or IS mode.

4. Transaction Ti can lock a node Q in X, SIX, or IX mode only if Ti currently has the parent of Q locked in either IX or SIX mode.

5. Transaction Ti can lock a node only if Ti has not previously unlocked any node (that is, Ti is two phase).

6. Transaction Ti can unlock a node Q only if Ti currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

Timestamp-Based Protocols

Timestamps:

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).

There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj .

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.

- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The Timestamp-Ordering Protocol: The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues read(Q).

a. If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction Ti issues write(Q).

a. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

b. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.

c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

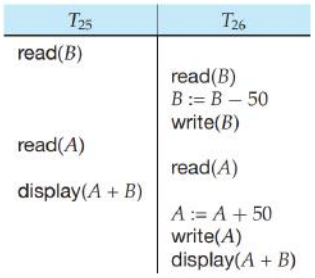
To illustrate this protocol, we consider transactions T25 and T26. Transaction T25 displays the contents of accounts A and B:

T_{25}: read(B);
read(A);
display($A + B$).

Transaction T_{26} transfers \$50 from account B to account A, and then displays the contents of both:

T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 15.17, TS(T25) < TS(T26), and the schedule is possible under the timestamp protocol.



T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figure 15.17 Schedule 3.

Thomas' Write Rule:

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this:

Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

Validation-Based Protocols

The validation protocol requires that each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

1. Read phase. During this phase, the system executes transaction Ti . It reads the values of the various data items and stores them in variables local to Ti . It performs all write operations on temporary local variables, without updates of the actual database.

2. Validation phase. The validation test (described below) is applied to transaction Ti . This determines whether Ti is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.

3. Write phase. If the validation test succeeds for transaction Ti , the temporary local variables that hold the results of any write operations performed by Ti are copied to the database. Read-only transactions omit this phase.

Each transaction must go through the phases in the order shown. However, phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction Ti :

1. Start(Ti), the time when Ti started its execution.

2. Validation(Ti), the time when Ti finished its read phase and started its validation phase.

3. Finish(Ti), the time when Ti finished its write phase.

The **validation test** for transaction Ti requires that, for all transactions Tk with TS(Tk) < TS(Ti), one of the following two conditions must hold:

1. Finish(Tk) < Start(Ti). Since Tk completes its execution before Ti started, the serializability order is indeed maintained.

2. The set of data items written by Tk does not intersect with the set of data items read by Ti , and Tk completes its write phase before Ti starts its validation phase (Start(Ti) < Finish(Tk) < Validation(Ti)). This condition ensures that the writes of Tk and Ti do not overlap. Since the writes of Tk do not affect the read of Ti , and since Ti cannot affect the read of Tk , the serializability order is indeed maintained.

As an illustration, consider again transactions T25 and T26. Suppose that TS(T25) < TS(T26). Then, the validation phase succeeds in the schedule 6 in Figure 15.19. Note that the writes to the actual variables are performed only after the validation phase of T26. Thus, T25 reads the old values of B and A, and this schedule is serializable.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

Figure 15.19 Schedule 6, a schedule produced by using validation.

Recovery System

An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a failure.

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. In this chapter, we shall consider only the following types of failure:

• **Transaction failure.** There are two types of errors that may cause a transaction to fail:

◦ **Logical error.** The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.

◦ **System error.** The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.

• **System crash.** There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

• **Disk failure.** A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

Recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

Storage

We identified three categories of storage:

• Volatile storage

• Nonvolatile storage

• Stable storage

Stable storage or, more accurately, an approximation thereof, plays a critical role in recovery algorithms.

Stable-Storage Implementation:

we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in:

• **Successful completion.** The transferred information arrived safely at its destination.

• **Partial failure.** A failure occurred in the midst of transfer, and the destination block has incorrect information.

• **Total failure.** The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a data-transfer failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.

2. When the first write completes successfully, write the same information onto the second physical block.

3. The output is completed only after the second write completes successfully.

Data Access:

The database is partitioned into fixed-length storage units called **blocks.** Blocks are the units of data transfer to and from disk, and may contain several data items.

Transactions input information from the **disk to main memory,** and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input(B) transfers the physical block B to main memory.

2. output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

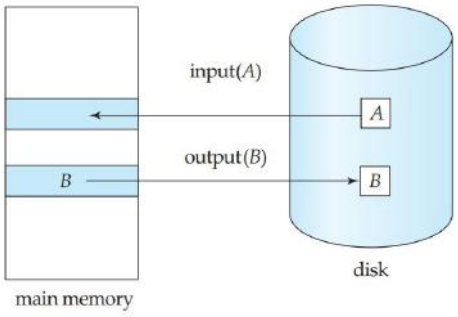


Figure 16.1 Block storage operations.

We transfer data by these two operations:

1. read(X) assigns the value of data item X to the local variable xi . It executes this operation as follows:

a. If block BX on which X resides is not in main memory, it issues input(BX).

b. It assigns to xi the value of X from the buffer block.

2. write(X) assigns the value of local variable xi to data item X in the buffer block. It executes this operation as follows:

a. If block BX on which X resides is not in main memory, it issues input(BX).

b. It assigns the value of xi to X in buffer BX.

Recovery and Atomicity

Our goal is to perform either all or no database modifications made by Ti . However, if Ti performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

Log Records:

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of log records. An **update log record** describes a single database write. It has these fields:

• **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.

• **Data-item identifier,** which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.

• **Old value**, which is the value of the data item prior to the write.

• **New value**, which is the value that the data item will have after the write.

We represent an update log record as , indicating that transaction Ti has performed a write on data item Xj . Xj had value V1 before the write, and has value V2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

• <Ti start>. Transaction Ti has started.

• <Ti commi0074>. Transaction Ti has committed.

• <Ti abort>. Transaction Ti has aborted.

Database Modification:

we need to consider the steps a transaction takes in modifying a data item:

1. The transaction performs some computations in its own private part of main memory.

2. The transaction modifies the data block in the disk buffer in main memory holding the data item.

3. The database system executes the output operation that writes the data block to disk.

This allows the system to perform undo and redo operations as appropriate.

• **Undo** using a log record sets the data item specified in the log record to the old value.

• **Redo** using a log record sets the data item specified in the log record to the new value.

Concurrency Control and Recovery:

If the concurrency control scheme allows a data item X that has been modified by a transaction T1 to be further modified by another transaction T2 before T1 commits, then undoing the effects of T1 by restoring the old value of X (before T1 updated X) would also undo the effects of T2. To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts.

Transaction Commit:

We say that a transaction has committed when its commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. If a system crash occurs before a log record < Ti commit> is output to stable storage, transaction Ti will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.

Using the Log to Redo and Undo Transactions:

Consider our simplified banking system. Let T0 be a transaction that transfers $50 from account A to account B:

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Figure 16.2 Portion of the system log corresponding to T_0 and T_1.

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let T_1 be a transaction that withdraws $100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction Ti , and their respective old and new values.

• redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

• undo(Ti) restores the value of all data items updated by transaction Ti to the old values.

Checkpoints:

a simple checkpoint scheme that (a) does not permit any updates to be performed while the checkpoint operation is in progress, and (b) outputs all modified buffer blocks to disk when the checkpoint is performed. We discuss later how to modify the checkpointing and recovery procedures to provide more flexibility by relaxing both these requirements.

A checkpoint is performed as follows:

1. Output onto stable storage all log records currently residing in main memory.

2. Output to the disk all modified buffer blocks.

3. Output onto stable storage a log record of the form < checkpoint L>, where L is a list of transactions active at the time of the checkpoint.

The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the <checkpoint L> record was written to the log. Let us denote this set of transactions as T.

• For all transactions Tk in T that have no <Tk commit> record or < Tk abort> record in the log, execute undo(Tk).

• For all transactions Tk in T such that either the record <Tk commit> or the record <Tk abort> appears in the log, execute redo(Tk).

Recovery Algorithm

Transaction Rollback:

First consider transaction rollback during normal operation (that is, not during recovery from a system crash). Rollback of a transaction Ti is performed as follows:

1. The log is scanned backward, and for each log record of Ti of the form <Ti, Xj, V1, V2> that is found:

a. The value V1 is written to data item Xj , and

b. A special redo-only log record Ti, Xj, V1> is written to the log, where V1 is the value being restored to data item Xj during the rollback. These log records are sometimes called compensation log records. Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.

2. Once the log record <Ti start> is found the backward scan is stopped, and a log record <Ti abort> is written to the log.

Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1. In the **redo phase**, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred. further, such incomplete transactions would have neither a nor a <Ti commit> record in the log.

The specific steps taken while scanning the log are as follows:

a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the <checkpoint L> log record.

b. Whenever a normal log record of the form <Ti, Xj, V1, V2>, or a redo-only log record of the form <Ti, Xj, V2> is encountered, the operation is redone; that is, the value V2 is written to data item Xj .

c. Whenever a log record of the form < Ti start> is found, Ti is added to undo-list.

d. Whenever a log record of the form <Ti abort> or is found, Ti is removed from undo-list.

2. In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

a. Whenever it finds a log record belonging to a transaction in the undo list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a <Ti start> log record for a transaction Ti in undo-list, it writes a <Ti abort> log record to the log, and removes Ti from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found <Ti start> log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

Buffer Management

Log-Record Buffering:

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

• Transaction Ti enters the commit state after the <Ti commit> log record has been output to stable storage.

• Before the <Ti commit> log record can be output to stable storage, all log records pertaining to transaction Ti must have been output to stable storage.

• Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.

This rule is called the **write-ahead logging (WAL)** rule.

Writing the buffered log to disk is sometimes referred to as a **log force**.

Database Buffering:

One might expect that transactions would force-output all modified blocks to disk when they commit. Such a policy is called the **force policy**. The alternative, the **no-force policy**, allows a transaction to commit even if it has modified some blocks that have not yet been written back to disk.

Similarly, one might expect that blocks modified by a transaction that is still active should not be written to disk. This policy is called the **no-steal policy**. The alternative, the **steal policy,** allows the system to write modified blocks to disk even if the transactions that made those modifications have not all committed.

Operating System Role in Buffer Management:

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements in Section 16.5.1, the operating system should not write out the database buffer pages itself, but instead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

Fuzzy Checkpointing:

The checkpointing technique described in Section 16.3.6 requires that all updates to the database be temporarily suspended while the checkpoint is in progress. If the number of pages in the buffer is large, a checkpoint may take a long time to finish, which can result in an unacceptable interruption in processing of transactions.

To avoid such interruptions, The checkpointing technique can be modified to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk. The checkpoint thus generated is a fuzzy checkpoint.

Failure with Loss of Nonvolatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure. In this section, we discuss only disk storage. Our discussions apply as well to other nonvolatile storage types.

The basic scheme is to dump the entire contents of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

One approach to database dumping requires that no transaction may be active during the dump procedure, and uses a procedure similar to checkpointing:

1. Output all log records currently residing in main memory onto stable storage.

2. Output all buffer blocks onto the disk.

3. Copy the contents of the database to stable storage.

4. Output a log record onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and checkpointing of buffers are similar.

Most database systems also support an **SQL dump**, which writes out SQL DDL statements and SQL insert statements to a file, which can then be reexecuted to re-create the database. Such dumps are useful when migrating data to a different instance of the database, or to a different version of the database software, since the physical locations and layout may be different in the other database instance or database software version.

Fuzzy dump schemes have been developed that allow transactions to be active while the dump is in progress. They are similar to fuzzy-checkpointing schemes; see the bibliographical notes for more details.

Early Lock Release and Logical Undo Operations

Logical Operations:

The insertion and deletion operations are examples of a class of operations that require logical undo operations since they release locks early; we call such operations logical operations. Such early lock release is important not only for indices, but also for operations on other system data structures that are accessed and updated very frequently; examples include data structures that track the blocks containing records of a relation, the free space in a block, and the free blocks in a database. If locks were not released early after performing operations on such data structures, transactions would tend to run serially, affecting system performance.

Logical Undo Log Records:

To allow logical undo of operations, before an operation is performed to modify an index, the transaction creates a log record <Ti, Oj, operation-begin>, where Oj is a unique identifier for the operation instance.5 While the system is executing the operation, it creates update log records in the normal fashion for all updates performed by the operation. Thus, the usual old-value and new-value information is written out as usual for each update performed by the operation; the old-value information is required in case the transaction needs to be rolled back before the operation completes. When the operation finishes, it writes an operation-end log record of the form <Ti, Oj, operation-end, U>, where the U denotes undo information.

For example, if the operation inserted an entry in a B+-tree, the undo information U would indicate that a deletion operation is to be performed, and would identify the B+-tree and what entry to delete from the tree. Such logging of information about operations is called logical

logging. In contrast, logging of old-value and new-value information is called physical logging, and the corresponding log records are called physical log records.

Data structures such as B+-trees would not be in a consistent state, and neither logical redo nor logical undo operations can be performed on an inconsistent data structure. To perform logical redo or undo, the database state on disk must be operation consistent, that is, it should not have partial effects of any operation.

An operation is said to be **idempotent** if executing it several times in a row gives the same result as executing it once. Operations such as inserting an entry into a B+-tree may not be idempotent, and the recovery algorithm must therefore make sure that an operation that has already been performed is not performed again.

Transaction Rollback With Logical Undo:

When rolling back a transaction Ti , the log is scanned backwards, and log records corresponding to Ti are processed as follows:

1. Physical log records encountered during the scan are handled as described earlier, except those that are skipped as described shortly. Incomplete logical operations are undone using the physical log records generated by the operation.

2. Completed logical operations, identified by operation-endrecords, are rolled back differently. Whenever the system finds a log record <Ti, Oj, operationend, U>, it takes special actions:

a. It rolls back the operation by using the undo information U in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed.

At the end of the operation rollback, instead of generating a log record <Ti, Oj , operation-end, U>, the database system generates a log record <Ti, Oj, operation-abort>.

b. As the backward scan of the log continues, the system skips all log records of transaction Ti until it finds the log record <Ti, Oj, operationbegin>. After it finds the operation-begin log record, it processes log records of transaction Ti in the normal manner again.

3. If the system finds a record <Ti, Oj , operation-abort>, it skips all preceding records (including the operation-end record for Oj) until it finds the record <Ti, Oj , operation-begin>.

4. As before, when the <Ti start> log record has been found, the transaction rollback is complete, and the system adds a record <Ti abort> to the log.'

Concurrency Issues in Logical Undo:

As mentioned earlier, it is important that the lower-level locks acquired during an operation are sufficient to perform a subsequent logical undo of the operation; otherwise concurrent operations that execute during normal processing may cause problems in the undo-phase. For example, suppose the logical undo of operation O1 of transaction T1 can conflict at the data item level with a concurrent operation O2 of transaction T2, and O1 completes while O2 does not. Assume also that neither

transaction had committed when the system crashed. The physical update log records of O2 may appear before and after the operation-end record for O1, and during recovery updates done during the logical undo of O1 may get fully or partially overwritten by old values during the physical undo of O2. This problem cannot occur if O1 had obtained all the lower-level locks required for the logical undo of O1, since then there cannot be such a concurrent O2.