RAJEEV GANDHI MEMORIAL COLLEGE OF ENGG.& TECH., NANDYAL-518 501 AUTONOMOUS COMPUTER SCIENCE AND ENGINEERING (A0514153) DATABASE MANAGEMENT SYSTEMS

OBJECTIVES:

- ❖ Advantages applications of DBMS and Database system structure.
- ❖ Schema design: ER model and conceptual design.
- ❖ Relational model and SQL basics.
- ❖ Relational algebra and Query optimization.
- ❖ Storage and efficient retrieval of data: various indexing techniques.
- ❖ Schema refinement: normalization and redundancy removal and functional dependant.
- ❖ Transaction management: locking protocols, serializability concepts etc.
- ❖ Concurrency control and crash recovery: various mechanisms, ARIES algorithm and deadlock concepts.

OUTCOMES:

- ➢ Students will learn about the need for DBMS, the largeness of the data and why it gives rise to steamϖ oriented processing and strategies and are at higher level than general purpose programming language such as JAVA.
- ➢ Students will learn about storage and efficient retrieval of large Information via algebraic query optimization and the use of indexing.
- ➢ Students will also learn basics of SQL and about primary key concepts and foreign key concepts.
- ➢ They will also learn about data manipulation (insertions deletions & updation) and triggers.
- ➢ Students will learn about functional dependency and the need for schema refinement (normalization) toϖ remove redundancy of data.
- ➢ Students will also learn about transaction management concurrency Control and crash recovery.

<u>CO-PO MAPPING:</u>

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	1		3										2		
CO2	1			3				2						2	3
CO3		2								1	1		1		
CO4							2								2
CO5			2				2							1	

UNIT I: Database System Applications, database System VS file System – View of Data – Data Abstraction –Instances and Schemas – data Models – the ER Model – Relational Model – Database Languages – DDL – DML – Database Access for applications Programs – Database Users and Administrator – Transaction Management – Database System Structure – Storage Manager – the Query Processor- Data base design and ER diagrams – Beyond ER Design- Entities, Attributes and Entity sets – Relationships and Relationship sets – Additional features of ER Model – Conceptual Design with the ER Model.

UNIT II: Introduction to the Relational Model – Integrity Constraint Over relations – Enforcing Integrity constraints – Querying relational data – Logical database Design – Introduction to Views – Destroying /altering Tables and Views. Relational Algebra – Selection and projection set operations – renaming – Joins – Division – Examples of Algebra queries – Relational calculus – Tuple relational Calculus – Domain relational calculus.

UNIT III: The Form of a Basic SQL Query – Examples of Basic SQL Queries – Introduction to Nested Queries – Correlated Nested Queries, Set – Comparison Operators – Aggregate Operators – NULL values – Comparison using Null values – Logical connectivity's – AND, OR and NOT – Impact on SQL Constructs – Outer Joins – Disallowing NULL values – Complex Integrity Constraints in SQL, Triggers and Active Data bases.

UNIT IV: Schema refinement – Problems Caused by redundancy – Decompositions – Problems related to decomposition – Functional dependencies-reasoning about FDS – FIRST, SECOND, THIRD Normal forms – BCNF – Lossless join Decomposition – Dependency preserving Decomposition – Schema refinement in Data base Design – Multi valued Dependencies – FORTH Normal Form.

UNIT V: Overview Of Transaction Management: The ACID Properties, Transactions and Schedules, Concurrent Execution of transactions-Lock Based Concurrency Control, Performance of Locking, Transaction Support in SQL. Concurrency Control: 2PL, Serializability and recoverability, Introduction Lock Management, Lock Conversions, Dealing with Deadlocks, Concurrency control without locking.

UNIT VI: Data on External Storage – File Organizations and Indexing – Cluster Indexes, Primary and Secondary Indexes – Index data Structures – Hash Based Indexing – Tree base Indexing – Comparison of File Organizations – The Memory Hierarchy, RAID, Disk Space Management, Buffer Manager, Files of Records, Page Formats, record Formats.

TEXT BOOKS:
1. Data base Management Systems, Raghu Ramakrishna, Johannes Gehrke, TATA McGraw Hill 3rd Edition
2. Data base System Concepts, Silberschatz, Korth, McGraw hill, V edition.
REFERENCES:
1. Data base Systems design, Implementation, and Management, Peter Rob & Carlos Coronel 7th Edition.
2. Fundamentals of Database Systems, ElmasriNavathe Pearson Education.
3. Introduction to Database Systems, C.J.Date Pearson Education

UNIT-I
INTRODUCTION TO DBMS

Database System Applications, **database System VS file System – View of Data – Data Abstraction** –Instances and Schemas – **data Models – the ER Model – Relational Model – Database Languages – DDL – DML –** Database Access for applications Programs – **Database Users and Administrator** – Transaction Management – **Database System Structure** – Storage Manager – the Query Processor-Application Architectures- History of Data base Systems. **Data base design and ER diagrams** – Beyond ER Design- Entities, Attributes and Entity sets – Relationships and Relationship sets – **Additional features of ER Model – Conceptual Design with the ER Model.**

Introduction:

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient manner*.
Managementof data involves both defining structures for storage of information and providingmechanisms for the manipulation of information. In addition, the databasesystem must ensure the safety of the information stored, despite system crashes orattempts at unauthorized access.

Database System Applications

Databases are widely used. Here are some representative applications:
• ***Banking***: For customer information, accounts, and loans, and banking transactions.
• ***Airlines***: For reservations and schedule information. Airlines were among thefirst to use databases in a geographically distributed manner—terminals situatedaround the world accessed the central database system through phonelines and other data networks.
• ***Universities***: For student information, course registrations, and grades.
• ***Credit card transactions***: For purchases on credit cards and generation of monthlystatements.
• ***Telecommunication***: For keeping records of calls made, generating monthly bills,maintaining balances on prepaid calling cards, and storing information aboutthe communication networks.
• ***Finance***: For storing information about holdings, sales, and purchases of financialinstruments such as stocks and bonds.
• ***Sales***: For customer, product, and purchase information.
• ***Manufacturing***: For management of supply chain and for tracking productionof items in factories, inventories of items in warehouses/stores, and orders foritems.
• ***Human resources***: For information about employees, salaries, payroll taxes andbenefits, and for generation of paychecks.

Database Systems versus File Systems

One way to keep the information on a computer isto store it in operating system files. To allow users to manipulate the information, thesystem has a number of application programs that manipulate the files, including
• A program to debit or credit an account
• A program to add a new account
• A program to find the balance of an account
• A program to generate monthly statements
System programmers wrote these application programs to meet the needs of thebank.New application programs are added to the system as the need arises. For example,suppose that the savings bank decides to offer checking accounts. As a result,the bank creates new permanent files that contain information about all the checkingaccounts maintained in the bank, and it may have to write new application programsto deal with situations that do not arise in savings accounts, such as overdrafts. Thus,as time goes by, the system acquires more files and more application programs.This typical **file-processing system** is supported by a conventional operating system.

The system stores permanent records in various files, and it needs differentapplication programs to extract records from, and add records to, the appropriatefiles. Before database management systems (DBMSs) came along, organizations usuallystored information in such systems.
Keeping organizational information in a file-processing system has a number ofmajor disadvantages:
• **Data redundancy and inconsistency**.
Since different programmers create thefiles and application programs over a long period, the various files are likelyto have different formats and the programs may be written in several programminglanguages. Moreover, the same information may be duplicated inseveral places (files). This redundancy leadsto 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**. Suppose that one of the bank officers needs tofind out the names of all customers who live within a particular postal-codearea. The officer asks the data-processing department to generate such a list.Because the designers of the original system did not anticipate this request,there is no application program on hand to meet it. There is, however, an applicationprogram to generate the list of *all* customers.
• **Data isolation**. Because data are scattered in various files, and files may be indifferent formats, writing new application programs to retrieve the appropriatedata is difficult.
• **Integrity problems**. The data values stored in the database must satisfy certaintypes of **consistency constraints**. Developers enforcethese constraints in the system by adding appropriate code in the various applicationprograms. However, when new constraints are added, it is difficultto change the programs to enforce them. The problem is compounded whenconstraints involve several data items from different files.
• **Atomicity problems**. A computer system, like any other mechanical or electricaldevice, is subject to failure. In many applications, it is crucial that, if afailure occurs, the data be restored to the consistent state that existed prior tothe failure.
• **Concurrent-access anomalies**. For the sake of overall performance of the systemand faster response, many systems allow multiple users to update thedata simultaneously. In such an environment, interaction of concurrent updatesmay result in inconsistent data.
• **Security problems**. Not every user of the database system should be able toaccess all the data.
These difficulties, among others, prompted the development of database systems.In what follows, we shall see the concepts and algorithms that enable database systemsto solve the problems with file-processing systems.

View of Data

A database system is a collection of interrelated files and a set of programs that allowusers to access and modify these files. A major purpose of a database system is toprovide users with an *abstract* view of the data. That is, the system hides certaindetails of how the data are stored and maintained.

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiencyhas led designers to use complex data structures to represent data in the database.Since many database-systems users are not computer trained, developers hide thecomplexity from users through several levels of abstraction, to simplify users' interactionswith the system:
• **Physical level**. The lowest level of abstraction describes *how* the data are actuallystored. The physical level describes complex low-level data structures indetail.
• **Logical level**. The next-higher level of abstraction describes *what* data arestored in the database, and what relationships exist among those data. Thelogical level thus describes the entire database in terms of a small numberof relatively simple structures. Although implementation of the simple structures

at the logical level may involve complex physical-level structures, theuser of the logical level does not need to be aware of this complexity.

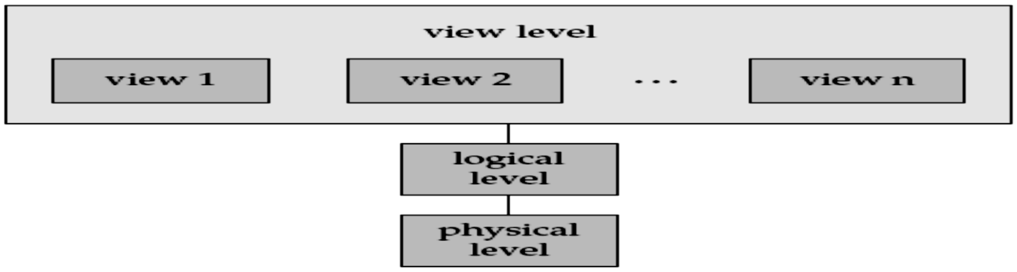


Figure 1.1 The three levels of data abstraction.

• **View level**. The highest level of abstraction describes only part of the entiredatabase. Even though the logical level uses simpler structures, complexityremains because of the variety of information stored in a large database. Manyusers of the database system do not need all this information; instead, theyneed to access only a part of the database. The view level of abstraction existsto simplify their interaction with the system. The system may provide manyviews for the same database.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection ofinformation stored in the database at a particular moment is called an **instance** of thedatabase. The overall design of the database is called the database **schema**.

The concept of database schemas and instances can be understood by analogy toa program written in a programming language. A database schema corresponds tothe variable declarations (along with associated type definitions) in a program. Eachvariable has a particular value at a given instant. The values of the variables in aprogram at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction.

The **physical schema** describes the database design at the physical level,while the **logical schema** describes the database design at the logical level. A databasemay also have several schemas at the view level, sometimes called **subschemas** thatdescribe different views of the database.

Data Models

The structure of a database is the **data model**: a collection of conceptualtools for describing data, data relationships, data semantics, and consistency constraints.

The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world thatconsists of a collection of basic objects, called *entities*, and of *relationships* among theseobjects. An entity is a "thing" or "object" in the real world that is distinguishablefrom other objects. For example, each person is an entity, and bank accounts can beconsidered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes*account-number* and *balance* may describe one particular account in a bank,and they form attributes of the *account* entity set. Similarly, attributes *customer-name*,*customer-street* address and *customer-city* may describe a *customer* entity.

A **relationship** is an association among several entities. For example, a *depositor*relationship associates a customer with each account that she has. The set of all entitiesof 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 graphicallyby an *E-R diagram*, which is built up from the following components:

• **Rectangles**, which represent entity sets
• **Ellipses**, which represent attributes
• **Diamonds**, which represent relationships among entity sets

• **Lines**, which link attributes to entity sets and entity sets to relationships
Each component is labeled with the entity or relationship that it represents.

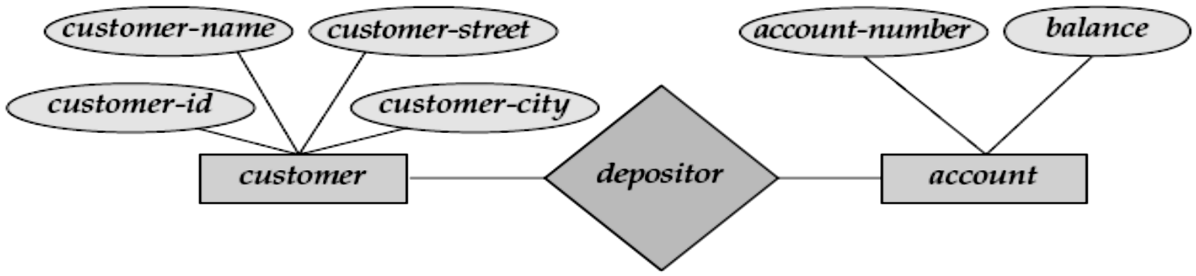


Figure 1.2 A sample E-R diagram.

Relational Model The relational model uses a collection of tables to represent both data and the relationshipsamong those data. Each table has multiple columns, and each column hasa unique name. A sample relational database comprising three tables:One shows details of bank customers, the second shows accounts, and the thirdshows which accounts belong to which customers. The relational model is an example of a record-based model. Record-based modelsare so named because the database is structured in fixed-format records of severaltypes. Each table contains records of a particular type. Each record type defines afixed number of fields, or attributes. The columns of the table correspond to the attributesof the record type.

Other Data Models

The **object-oriented data model** is another data model that has seen increasing attention.The object-oriented model can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Chapter 8 examines theobject-oriented data model.

The **object-relational data model** combines features of the object-oriented datamodel and relational data model. Historically, two other data models, the **network data model** and the **hierarchicaldata model**, preceded the relational data model. These models were tied closely tothe underlying implementation, and complicated the task of modeling data.

Database Languages

A database system provides a **data definition language** to specify the database schemaand a **data manipulation language** to express database queries and updates. Inpractice, the data definition and data manipulation languages are not two separatelanguages; instead they simply form parts of a single database language, such as thewidely used SQL language.

Data-Definition Language

We specify a database schema by a set of definitions expressed by a special languagecalled a **data-definition language (DDL)**.

For instance, the following statement in the SQL language defines the *account* table:

create table *account*(*account-number* **char**(10),*balance* **integer**)

Execution of the above DDL statement creates the *account* table. In addition, it updatesa special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a tableis an example of metadata. A database system consults the data dictionary beforereading or modifying actual data.We specify the storage structure and access methods used by the database systemby a set of statements in a special type of DDL called a **data storage and definition** language.These statements define the implementation details of the database schemas,which are usually hidden from the users.The data values stored in the database must satisfy certain **consistency constraints**.

Data-Manipulation Language

Data manipulation is
• The retrieval of information stored in the database
• The insertion of new information into the database
• The deletion of information from the database
• The modification of information stored in the database
A **data-manipulation language (DML)** is a language that enables users to accessor manipulate data as organized by the appropriate data model. There are basicallytwo types:
• **Procedural DMLs** require a user to specify *what* data are needed and *how* toget those data.

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

Declarative DMLs are usually easier to learn and use than are procedural DMLs.However, since a user does not have to specify how to get the data, the databasesystem has to figure out an efficient means of accessing data. The DML component ofthe SQL language is nonprocedural.
A **query** is a statement requesting the retrieval of information. The portion of aDML that involves information retrieval is called a **query language**.
This query in the SQL language finds the name of the customer whose customer-idis 192-83-7465:
Eg:
select*customer. Customer-name***from** *customer***where** *customer. Customer-id*= 192-83-7465

Database Access from Application Programs

Application programs are programs that are used to interact with the database. Applicationprograms are usually written in a *host* language, such as COBOL, C, C++, orJava.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 canbe used to send DML and DDL statements to the database, and retrieve theresults.The Open Database Connectivity (ODBC) standard defined by Microsoftfor use with the C language is a commonly used application program interfacestandard. The Java Database Connectivity (JDBC) standard provides correspondingfeatures to the Java language.
• By extending the host language syntax to embed DML calls within the hostlanguage program. Usually, a special character prefaces DML calls, and a preprocessor,called the *DML* **precompiler**, converts the DML statements to normalprocedure calls in the host language.

Database Users and Administrators

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

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the waythey expect to interact with the system. Different types of user interfaces have beendesigned for the different types of users.
• **Naive users** are unsophisticated users who interact with the system by invokingone of the application programs that have been written previously.
• **Application programmers** are computer professionals who write applicationprograms. Application programmers can choose from many tools to developuser interfaces. **Rapid application development (RAD)** tools are tools that enablean application programmer to construct forms and reports without writinga program. There are also special types of programming languages thatcombine imperative control structures (for example, for loops, while loopsand if-then-else statements) with statements of

the data manipulation language.These languages, sometimes called *fourth-generation languages*, ofteninclude special features to facilitate the generation of forms and the display ofdata on the screen.
• **Sophisticated users** interact with the system without writing programs. Instead,they form their requests in a database query language. They submiteach such query to a **query processor**, whose function is to break down DMLstatements into instructions that the storage manager understands. Analystswho submit queries to explore data in the database fall in this category. **Online analytical processing (OLAP)** tools simplify analysts' tasks by lettingthem view summaries of data in different ways.
• **Specialized users** are sophisticated users who write specialized databaseapplications 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.

Database Administrator

One of the main reasons for using DBMSs is to have central control of both the dataand the programs that access those data. A person who has such central control overthe system is called a **database administrator** (**DBA**). The functions of a DBA include:
• **Schema definition**. The DBA creates the original database schema by executinga set of data definition statements in the DDL.
• **Storage structure and access-method definition**.
• **Schema and physical-organization modification**. The DBA carries out changesto the schema and physical organization to reflect the changing needs of theorganization, or to alter the physical organization to improve performance.
• **Granting of authorization for data access**. By granting different types ofauthorization, the database administrator can regulate which parts of the databasevarious users can access. The authorization information is kept in aspecial system structure that the database system consults whenever someoneattempts to access the data in the system.
• **Routine maintenance**. Examples of the database administrator's routinemaintenance activities are:
periodically backing up the database, either onto tapes or onto remoteservers, to prevent loss of data in case of disasters such as flooding.
▫Ensuring that enough free disk space is available for normal operationsand upgrading disk space as required.
▫Monitoring jobs running on the database and ensuring that performanceis not degraded by very expensive tasks submitted by some users.

Transaction Management

Often, several operations on the database form a single logical unit of work. An exampleis a funds transfer, as in Section 1.2, in which one account (say *A*) is debited andanother account (says*B*) is credited. Clearly, it is essential that either both the creditand debit occur, or that neither occur. That is, the funds transfer must happen in itsentirety or not at all. This all-or-none requirement is called **atomicity**. In addition, itis essential that the execution of the funds transfer preserve the consistency of thedatabase. That is, the value of the sum $A + B$ must be preserved. This correctnessrequirement is called **consistency**. Finally, after the successful execution of a fundstransfer, the new values of accounts *A* and *B* must persist, despite the possibility ofsystem failure. This persistence requirement is called **durability**.
A **transaction** is a collection of operations that performs a single logical functionin a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistencyconstraints. That is, if the database was consistent when a transaction started, thedatabase must be consistent when the transaction successfully terminates.

Ensuring the atomicity and durability properties is the responsibility of the databasesystem itself—specifically, of the **transaction-management component**. In theabsence of failures, all transactions complete successfully, and atomicity is achievedeasily.
Finally, when several transactions update the database concurrently, the consistencyof data may no longer be preserved, even though each individual transactionis correct. It is the responsibility of the **concurrency-control manager** to controlthe interaction among the concurrent transactions, to ensure the consistency of thedatabase.
Database systems designed for use on small personal computers may not haveall these features.

Database System Structure

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 bebroadly divided into the storage manager and the query processor components.
The storage manager is important because databases typically require a largeamount of storage space. The query processor is important because it helps the database system simplifyand facilitate access to data. High-level views help to achieve this goal; with them,users of the system are not be burdened unnecessarily with the physical details of theimplementation of the system. However, quick processing of updates and queriesis important. It is the job of the database system to translate updates and querieswritten in a nonprocedural language, at the logical level, into an efficient sequence ofoperations at the physical level.

Storage Manager

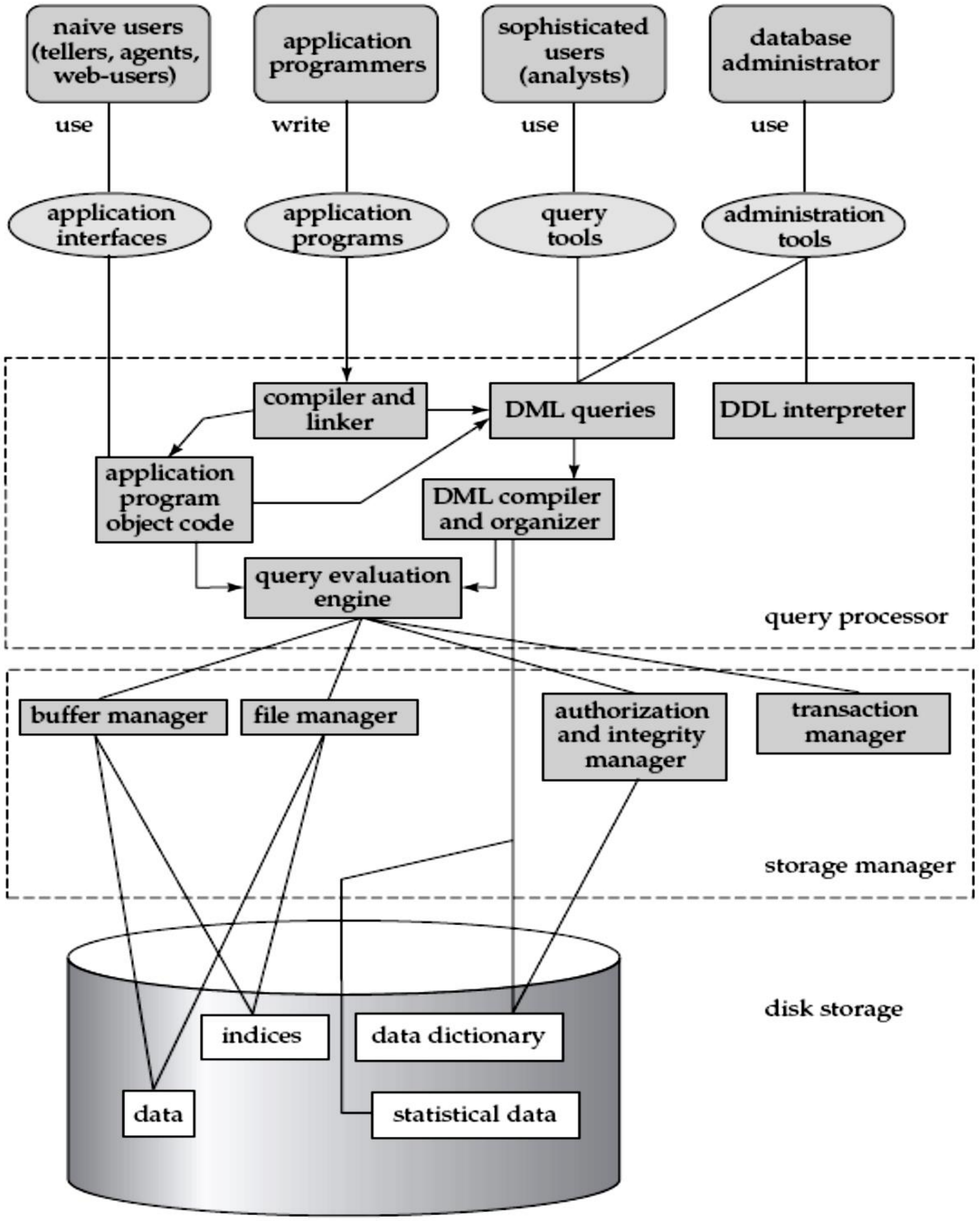
A *storage manager* is a program module that provides the interface between the low-leveldata stored in the database and the application programs and queries submittedto the system. The storage manager is responsible for the interaction with the filemanager. The raw data are stored on the disk using the file system, which is usuallyprovided by a conventional operating system. The storage manager translatesthe various DML statements into low-level file-system commands. Thus, the storagemanager is responsible for storing, retrieving, and updating data in the database.The storage manager components include:
• **Authorization and integrity manager**, which tests for the satisfaction of integrityconstraints and checks the authority of users to access data.
• **Transaction manager**, which ensures that the database remains in a consistent(correct) state despite system failures, and that concurrent transaction executionsproceed without conflicting.
• **File manager**, which manages the allocation of space on disk storage and thedata structures used to represent information stored on disk.
• **Buffer manager**, which is responsible for fetching data from disk storage intomain memory, and deciding what data to cache in main memory. The buffermanager is a critical part of the database system, since it enables the databaseto handle data sizes that are much larger than the size of main memory.The storage manager implements several data structures as part of the physicalsystem implementation:
• **Data files**, which store the database itself.
• **Data dictionary**, which stores metadata about the structure of the database, inparticular the schema of the database.
• **Indices**, which provide fast access to data items that hold particular values.

The Query Processor

The query processor components include
• **DDL interpreter**, which interprets DDL statements and records the definitionsin the data dictionary.
• **DML** compiler, which translates DML statements in a query language into anevaluation plan consisting of low-level instructions that the query evaluationengine understands.

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

• **Query evaluation engine**, which executes low-level instructions generated bythe DML compiler. Figure 1.4 shows these components and the connections among them.



The DBMS accepts SQL commands generated from a variety of user interfaces, producesquery evaluation plans, executes these plans against the database, and returnsthe answers.

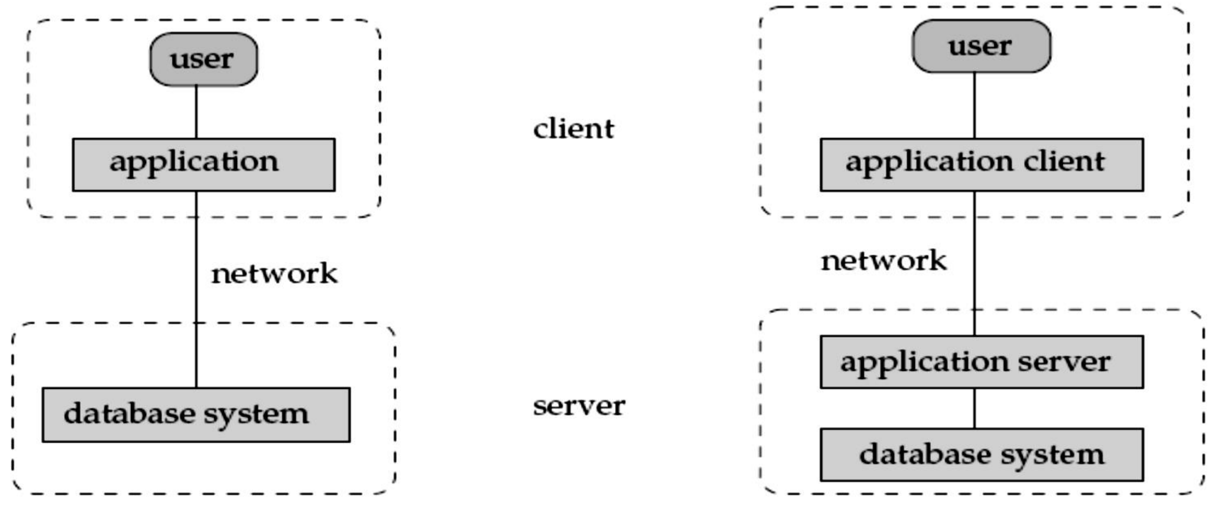
When a user issues a query, the parsed query is presented to a**query optimizer**, whichuses information about how the data is stored to produce an efficient execution planfor evaluating the query.

An**execution plan** is a blueprint for evaluating a query, andis usually represented as a tree of relational operators (with annotations that containadditional detailed information about which access methods to use, etc.). Relational operators serve as the building blocksfor evaluating queries posed against the data.
The code that implements relational operators sits on top of the file and access methodslayer. This layer includes a variety of software for supporting the concept of a**file**,which, in a DBMS, is a collection of pages or a collection of records. This layer typicallysupports a **heap file**, or file of unordered pages, as well as indexes. In addition tokeeping track of the pages in a file, this layer organizes the information within a page.
The files and access methods layer code sits on top of the**buffer manager**, whichbrings pages in from disk to main memory as needed in response to read requests.Buffer management is discussed inChapter 7.The lowest layer of the DBMS software deals with management of space on disk, wherethe data is stored. Higher layers allocate, de-allocate, read, and write pages through(routines provided by) this layer, called the**disk space manager**. This layer isdiscussed in Chapter 7.
The DBMS supports concurrency and crash recovery by carefully scheduling user requestsand maintaining a log of all changes to the database. DBMS components associatedwith concurrency control and recovery include the**transaction manager**, whichensures that transactions request and release locks according to a suitable locking protocoland schedules the execution transactions; the**lock manager**, which keeps trackof requests for locks and grants locks on database objects when they become available;and the recovery manager, which is responsible for maintaining a log, and restoringthe system to a consistent state after a crash. The disk space manager, buffer manager,and file and access method layers must interact with these components. We discussconcurrency control and recovery in detail in Chapter 18.

Application Architectures

Most users of a database system today are not present at the site of the databasesystem, but connect to it through a network. We can therefore differentiate betweenclient machines, on which remote database users work, and server machines, onwhich the database system runs.
Database applications are usually partitioned into two or three parts, as in Figure1.5. In a **two-tier architecture**, the application is partitioned into a componentthat resides at the client machine, which invokes database system functionality at theserver machine through query language statements. Application program interfacestandards like ODBC and JDBC are used for interaction between the client and theserver.

In contrast, in **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.



a. two-tier architecture b. three-tier architecture

Three-tier applications are more appropriate for large applications, and forapplications that run on the World Wide Web.

History of Database Systems

Data processing drives the growth of computers, as it has from the earliest days ofcommercial computers. In fact, automation of data processing tasks predates computers.Punched cards, invented by Hollerith, were used at the very beginning of thetwentieth century to record U.S. census data, and mechanical systems were used to process the cards and tabulate results. Punched cards were later widely used as ameans of entering data into computers.

Techniques for data storage and processing have evolved over the years:

·**1950s and early 1960s**: Magnetic tapes were developed for data storage. Dataprocessing tasks such as payroll were automated, with data stored on tapes.Processing of data consisted of reading data from one or more tapes and writing data to a new tape. Data could also be input from punched card decks,and output to printers.

·**Late 1960s and 1970s**:Widespread use of hard disks in the late 1960s changedthe scenario for data processing greatly, since hard disks allowed direct accessto data. The position of data on disk was immaterial, since any location on diskcould be accessed in just tens of milliseconds.

·**1980s**: Although academically interesting, the relational model was not usedin practice initially, because of its perceived performance disadvantages; relationaldatabases could not match the performance of existing network andhierarchical databases. That changed with System R, a groundbreaking projectat IBM Research that developed techniques for the construction of an efficientrelational database system.

·**Early 1990s**: The SQL language was designed primarily for decision supportapplications, which are query intensive, yet the mainstay of databases in the1980s was transaction processing applications, which are update intensive.Decision support and querying re-emerged as a major application area fordatabases.

·**Late 1990s**: The major event was the explosive growth of theWorldWideWeb.

Database design and ER model

Databases were deployed much more extensively than ever before. Databasesystems also had to supportWeb interfaces to data.

The database design process can be divided into six steps. The ER model is mostrelevant to the first three steps:

(1) Requirements Analysis: The very first step in designing a database applicationis to understand what data is to be stored in the database, what applications must bebuilt on top of it, and what operations are most frequent and subject to performancerequirements. In other words, we must find out what the users want from the database.This is usually an informal process that involves discussions with user groups, a studyof the current operating environment and how it is expected to change, analysis ofany available documentation on existing applications that are expected to be replacedor complemented by the database, and so on. Several methodologies have been proposedfor organizing and presenting the information gathered in this step, and someautomated tools have been developed to support this process.

(2) Conceptual Database Design: The information gathered in the requirementsanalysis step is used to develop a high-level description of the data to be stored in thedatabase, along with the constraints that are known to hold over this data. This stepis often carried out using the ER model, or a similar high-level data model, and isdiscussed in the rest of this chapter.

(3) Logical Database Design: We must choose a DBMS to implement our databasedesign, and convert the conceptual database design into a database schema in the datamodel of the chosen DBMS. We will only consider relational DBMSs, and therefore,the task in the logical design step is to convert an ER schema into a relational databaseschema. We discuss this step in detail in Chapter 3; the result is a conceptual schema,sometimes called the**logical schema**, in the relational data model.

The entity-relationship (ER) data model allows us to describe the data involved in areal-world enterprise in terms of objects and their relationships and is widely used todevelop an initial database design. The ER model is important primarily for its role in database design. It provides usefulconcepts that allow us to move from an informal description of what users want fromtheir database to a more detailed and precise, description that can be implementedin a DBMS.

Beyond the ER Model

ER modeling is sometimes regarded as a complete approach to designing a logicaldatabase schema. This is incorrect because the ER diagram is just an approximatedescription of the data, constructed through a very subjective evaluation of the informationcollected during requirements analysis. A more careful analysis can often refinethe logical schema obtained at the end of Step 3. Once we have a good logical schema,we must consider performance criteria and design the physical schema. Finally, wemust address security issues and ensure that users are able to access the data theyneed, but not data that we wish to hide from them. The remaining three steps ofdatabase design are briefly described below:

(4) Schema Refinement: The fourth step in database design is to analyze thecollection of relations in our relational database schema to identify potential problems,and to refine it. In contrast to the requirements analysis and conceptual design steps,which are essentially subjective, schema refinement can be guided by some elegant andpowerful theory. We discuss the theory of normalizing relations|restructuring themto ensure some desirable properties.

(5) Physical Database Design: In this step we must consider typical expectedworkloads that our database must support and further refine the database design toensure that it meets desired performance criteria. This step may simply involve buildingindexes on some tables and clustering some tables, or it may involve a substantialredesign of parts of the database schema obtained from the earlier design steps.

(6) Security Design: In this step, we identify different user groups and differentroles played by various users (e.g., the development team for a product, the customersupport representatives, the product manager). For each role and user group, we mustidentify the parts of the database that they must be able to access and the parts of thedatabase that they should not be allowed to access, and take steps to ensure that theycan access only the necessary parts. A DBMS provides several mechanisms to assist in this step.

ENTITIES, ATTRIBUTES, AND ENTITY SETS

An**entity** is an object in the real world that is distinguishable from other objects. It is often useful to identify a collection of similar entities. Such a collection iscalled an**entity set**. Note that entity sets need not be disjoint; An entity is described using a set of**attributes**. All entities in a given entity set havethe same attributes; this is essentially what we mean by similar. For each attribute associated with an entity set, we must identify a domain of possiblevalues, for each entity set, we choose a key.

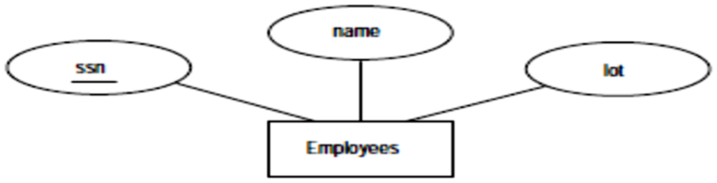


Figure 2.1 The Employees Entity Set

A **key** is a minimal set of attributes whose values uniquely identify an entity in theset. There could be more than one**candidate** key; if so, we designate one of them asthe**primary** key. For now we will assume that each entity set contains at least oneset of attributes that uniquely identifies an entity in the entity set; that is, the set ofattributes contains a key.

RELATIONSHIPS AND RELATIONSHIP SETS

A**relationship** is an association among two or more entities. For example, we mayhave the relationship that Attishoo works in the pharmacy department. As withentities, we may wish to collect a set of similar relationships into a**relationship set**.

A relationship set can be thought of as a set of n-tuples:

$$\{(e1, \ldots\ldots en) \mid e1 \in E1, \ldots\ldots en \in En\}$$

Each n-tuple denotes a relationship involving n entities e1 through en, where entity ei is in entity set Ei.

In Figure 2.2 we show the relationship set Works In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

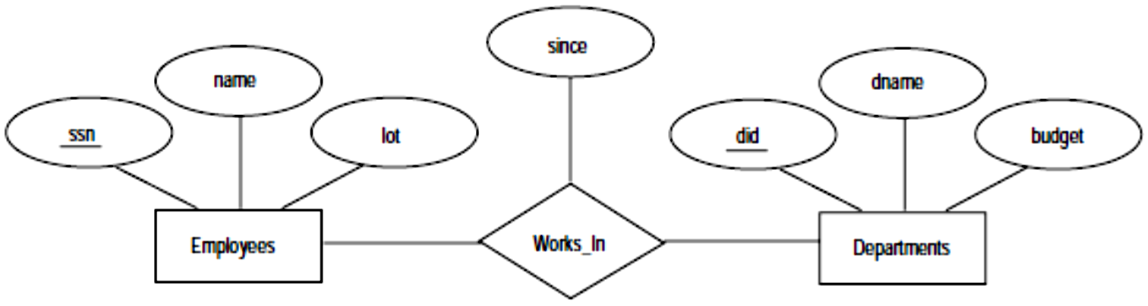


Figure 2.2 The Works_In Relationship Set

A relationship can also have**descriptive attributes**. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in Figure 2.2by adding an attribute, since, to Works In. A relationship must be uniquely identified
by the participating entities, without reference to the descriptive attributes. In theWorks In relationship set, for example, each Works In relationship must be uniquelyidentified by the combination of employee ssn and department did. Thus, for a givenemployee-department pair, we cannot have more than one associated since value.An**instance** of a relationship set is a set of relationships. Intuitively, an instancecan be thought of as a ‘snapshot' of the relationship set at some instant in time.
Each Employees entity is denoted by its ssn, and each Departments entity is denoted by its did, for simplicity.
The since value is shown beside each relationship.

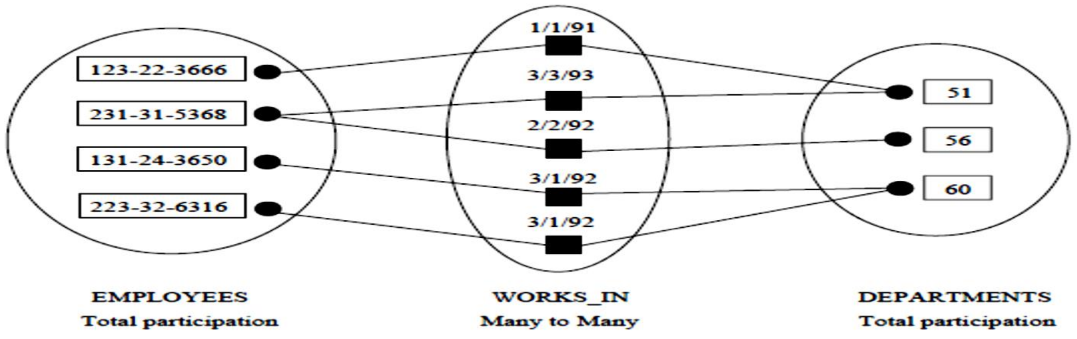


Figure 2.3 An Instance of the Works_In Relationship Set

As another example of an ER diagram, suppose that each department has offices in several locations and we want to record the locations at which each employee works. This relationship is**ternary** because we must record an association between an employee, a department, and a location.

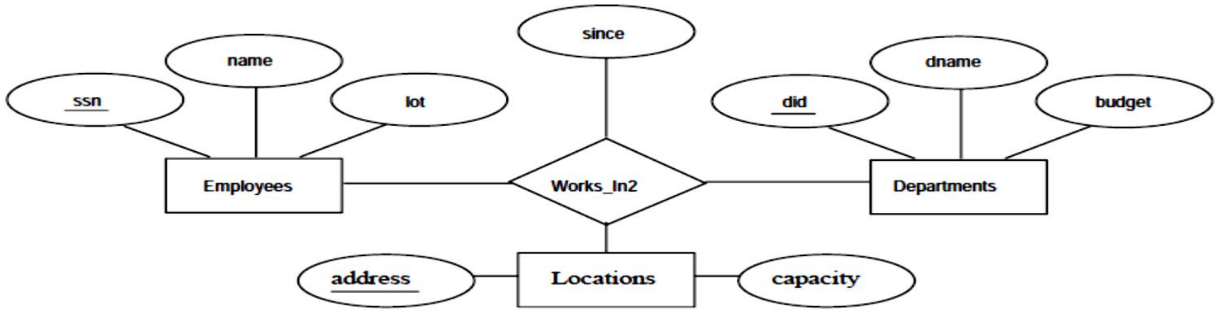


Figure 2.4 A Ternary Relationship Set

ADDITIONAL FEATURES OF THE ER MODEL

We now look at some of the constructs in the ER model that allow us to describe somesubtle properties of the data. The expressiveness of the ER model is a big reason forits widespread use.

Key Constraints

Consider the Works In relationship shown in Figure 2.2. An employee can work inseveral departments, and a department can have several employees, as illustrated inthe Works In instance shown in Figure 2.3. Employee 231-31-5368 has worked inDepartment 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 hastwo employees.

Now consider another relationship set called Manages between the Employees and Departmentsentity sets such that each department has at most one manager, although asingle employee is allowed to manage more than one department. The restriction thateach department has at most one manager is an example of a**key constraint**, andit implies that each Departments entity appears in at most one Manages relationship in any allowable instance of Manages. This restriction is indicated in the ER diagramof Figure 2.6 by using an arrow from Departments to Manages. Intuitively, the arrowstates that given a Departments entity, we can uniquely determine the Managesrelationship in which it appears.

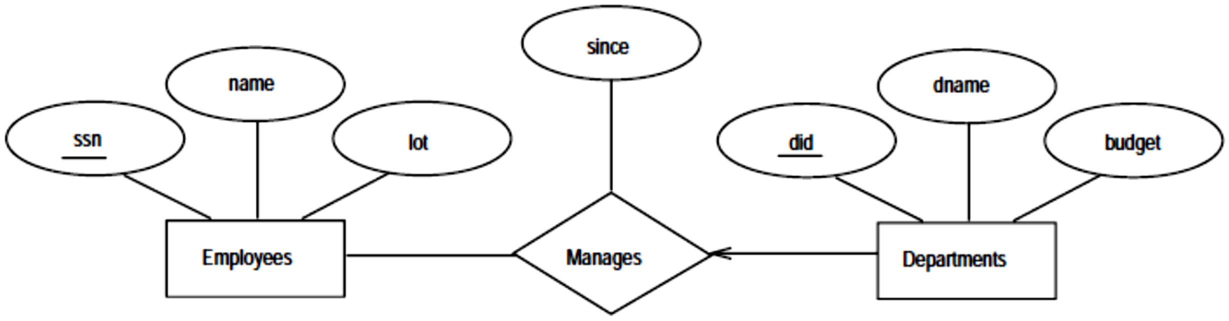


Figure 2.6 Key Constraint on Manages

An instance of the Manages relationship set is shown in Figure 2.7. While this is alsoa potential instance for the Works In relationship set, the instance of Works In shownin Figure 2.3 violates the key constraint on Manages.

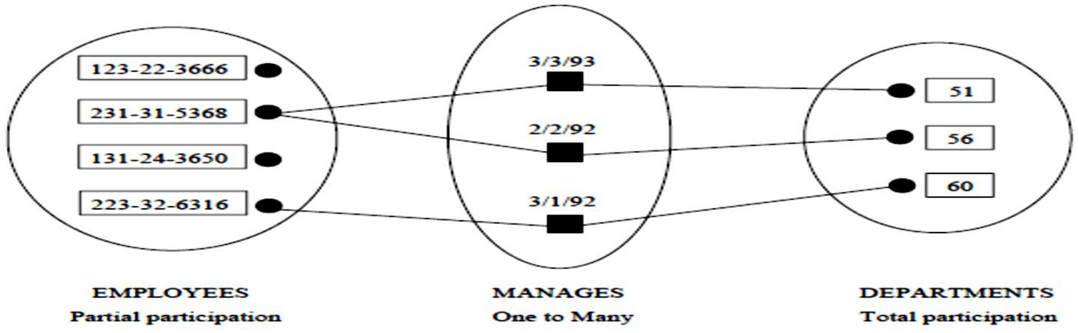


Figure 2.7 An Instance of the Manages Relationship Set

A relationship set like Manages is sometimes said to be**one-to-many**, to indicate thatone employee can be associated with many departments (in the capacity of a manager),whereas each department can be associated with at most one employee as its manager.In contrast, the Works In relationship set, in which an employee is allowed to work inseveral departments and a department is allowed to have several employees, is said tobe**many-to-many**.

Key Constraints for Ternary Relationships

We can extend this conventionand the underlying key constraint conceptto relationshipsets involving three or more entity sets: If an entity set E has a key constraintin a relationship set R, each entity in an instance of E appears in at most one relationshipin (a corresponding instance of) R. To indicate a key constraint on entity setE in relationship set R, we draw an arrow from E to R.In Figure 2.8, we show a ternary relationship with key constraints. Each employeeworks in at most one department, and at a single location.

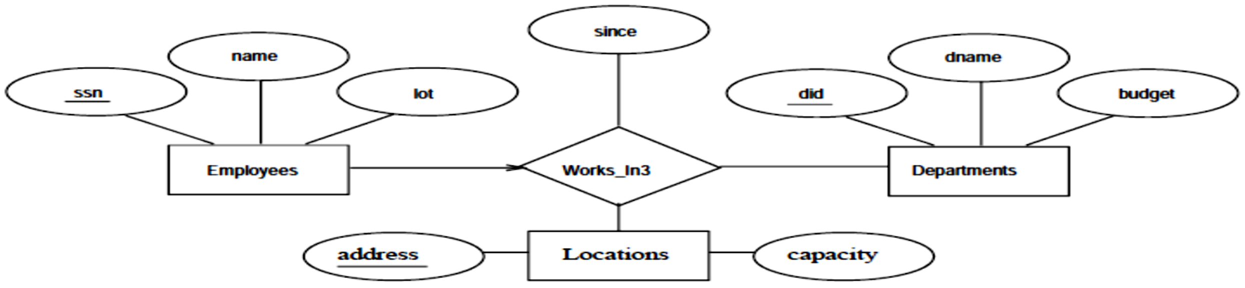


Figure 2.8 A Ternary Relationship Set with Key Constraints

An instance of theWorks In3 relationship set is shown in Figure 2.9. Notice that each department can beassociated with several employees and locations, and each location can be associatedwith several departments and employees; however, each employee is associated with asingle department and location.

Participation Constraints

The key constraint on Manages tells us that a department has at most one manager.A natural question to ask is whether every department has a manager. Let us say thatevery department is required to have a manager. This requirement is an example ofa**participation constraint**; the participation of the entity set Departments in therelationship set Manages is said to be**total**. A participation that is not total is said tobe**partial**. As an example, the participation of the entity set Employees in Managesis partial, since not every employee gets to manage a department. Revisiting the Works In relationship set, it is natural to expect that each employeeworks in at least one department and that each department has at least one employee.
This means that the participation of both Employees and Departments in Works Inis total. The ER diagram in Figure 2.10 shows both the Manages and Works Inrelationship sets and all the given constraints. If the participation of an entity setin a relationship set is total, the two are connected by a thick line; independently,the presence of an arrow indicates a key constraint. The instances of Works In andManages shown in Figures 2.3 and 2.7 satisfy all the constraints in Figure 2.10.

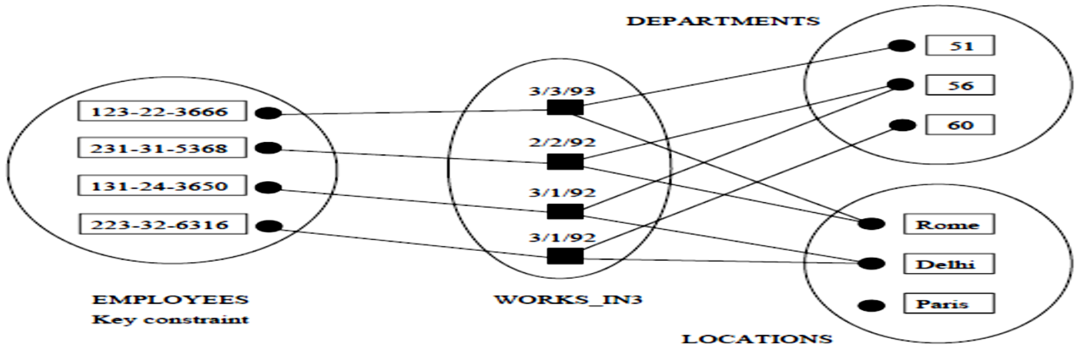


Figure 2.9 An Instance of Works_In3

Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include akey. This assumption does not always hold. For example, suppose that employees canpurchase insurance policies to cover their dependents. We wish to record informationabout policies, including who is

covered by each policy, but this information is reallyyour only interest in the dependents of an employee. If an employee quits, any policyowned by the employee is terminated and we want to delete all the relevant policy anddependent information from the database.
We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thusthe attributes of the Dependents entity set might be pname and age. The attributepname does not identify a dependent uniquely.

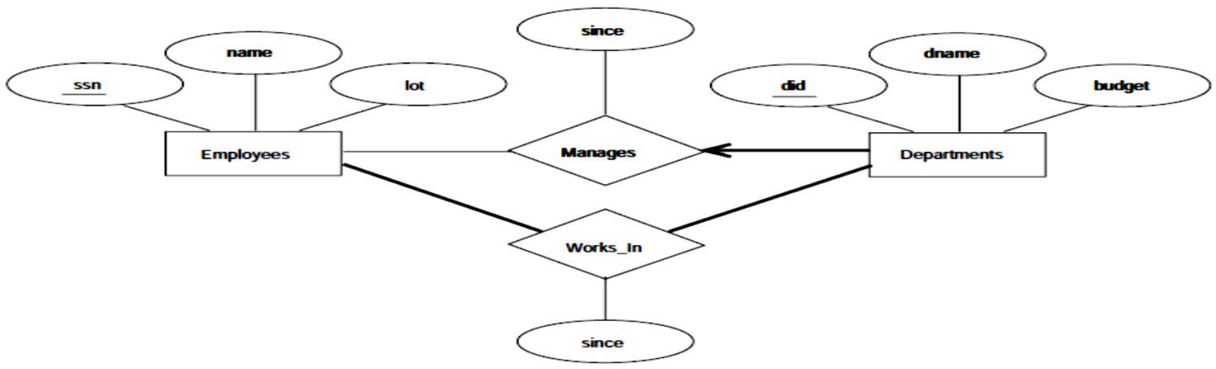


Figure 2.10 Manages and Works_In

Dependents is an example of a**weak entity set**. A weak entity can be identifieduniquely only by considering some of its attributes in conjunction with the primarykey of another entity, which is called the**identifying owner**.The following restrictions must hold:
The owner entity set and the weak entity set must participate in a one-to-manyrelationship set (one owner entity is associated with one or more weak entities,but each weak entity has a single owner). This relationship set is called the identifying **relationship set** of the weak entity set.The weak entity set must have total participation in the identifying relationshipset.

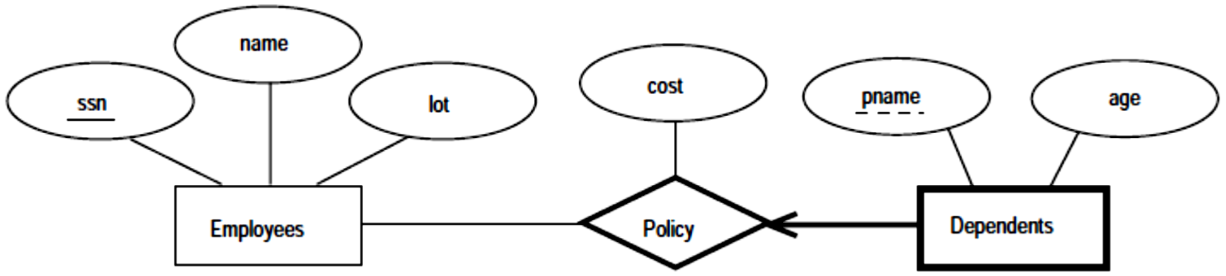


Figure 2.11 A Weak Entity Set

Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. Forexample, we might want to talk about an Hourly Emps entity set and a Contract Empsentity set to distinguish the basis on which they are paid. We might have attributeshours worked and hourly wage defined for Hourly Emps and an attribute contracted defined for Contract Emps.
We want the semantics that every entity in one of these sets is also an Employees entity,and as such must have all of the attributes of Employees defined. Thus, the attributesdefined for an Hourly Emps entity are the attributes for Employees plus Hourly Emps.We say that the attributes for the entity set Employees are**inherited** by the entityset Hourly Emps, and that Hourly Emps ISA (read is a)

Employees. In addition|and in contrast to class hierarchies in programming languages such as C++|there isa constraint on queries over instances of these entity sets: A query that asks for allEmployees entities must consider all Hourly Emps

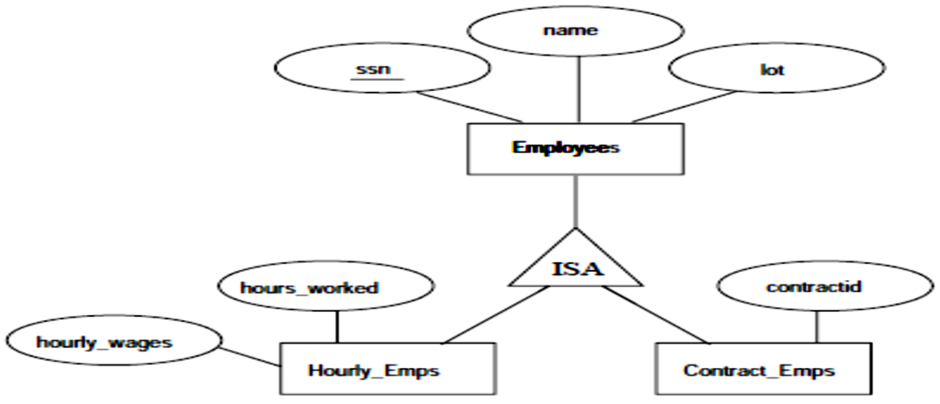


Figure 2.12 Class Hierarchy

A class hierarchy can be viewed in one of two ways:
Employees is specialized into subclasses. **Specialization** is the process of identifying subsets of an entity set (the superclass) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added. Hourly Emps and Contract Emps are generalized by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor Vehicles. **Generalization** consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.
We can specify two kinds of constraints with respect to ISA hierarchies, namely, overlap and coveringconstraints.**Overlap constraints**determine whether two subclasses are allowed to contain the same entity.
Covering constraintsdetermine whether the entities in the subclasses collectively include all entities in the superclass.

Aggregation

As we have defined it thus far, a relationship set is an association between entity sets.Sometimes we have to model a relationship between a collection of entities and relationships.Suppose that we have an entity set called Projects and that each Projectsentity is sponsored by one or more departments. The Sponsors relationship set capturesthis information. A department that sponsors a project might assign employeesto monitor the sponsorship. Intuitively, Monitors should be a relationship set thatassociates a Sponsors relationship (rather than a Projects or Departments entity) withan Employees entity. However, we have defined relationships to associate two or moreentities. In order to define a relationship set such as Monitors, we introduce a new feature of theER model, called aggregation. **Aggregation** allows us to indicate that a relationshipset (identified through a dashed box) participates in another relationship set.

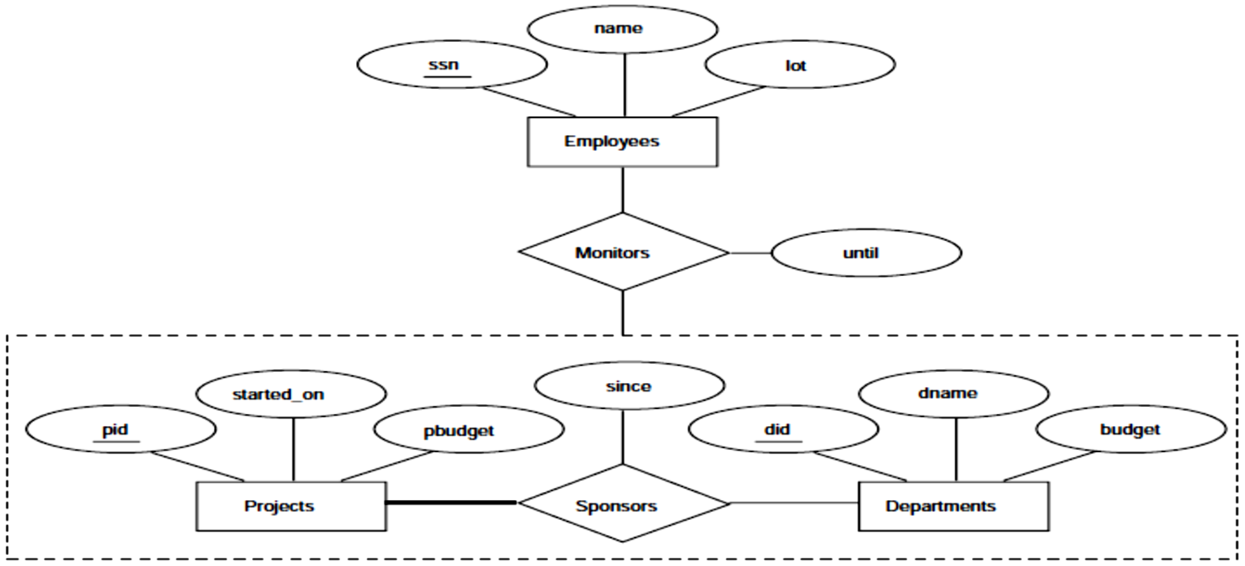


Figure 2.13 Aggregation

CONCEPTUAL DATABASE DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we usebinary or ternary relationships?
- Should we use aggregation?

Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether aproperty should be modeled as an attribute or as an entity set (and related to the firstentity set using a relationship set). For example, consider adding address informationto the Employees entity set. One option is to use an attribute address. This option isappropriate if we need to record only one address per employee, and it suffixes to thinkof an address as a string. An alternative is to create an entity set called Addressesand to record associations between employees and addresses using a relationship (say,Has Address). This more complex alternative is necessary in two situations:We have to record more than one address for an employee.We want to capture the structure of an address in our ER diagram. For example,we might break down an address into city, state, country, and Zip code, in additionto a string for street information. By representing an address as an entity withthese attributes, we can support queries such as \Find all employees with anaddress in Madison, WI."

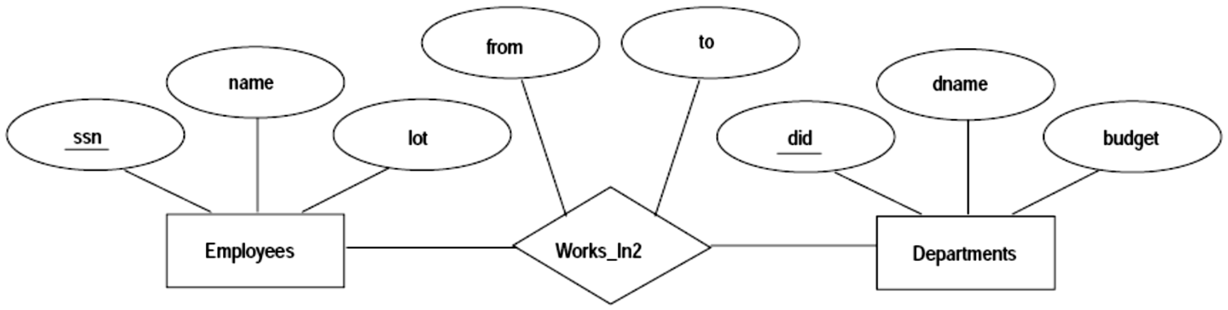


Figure 2.14 The Works_In2 Relationship Set

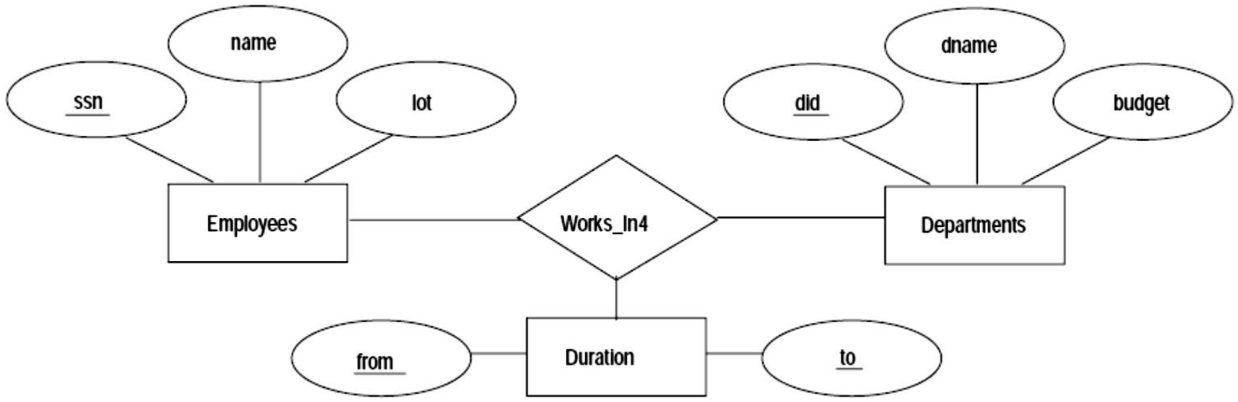


Figure 2.15 The Works_In4 Relationship Set

Entity versus Relationship

Consider the relationship set called Manages, suppose that each department manager is given a discretionary budget (dbudget), as shown in Figure 2.16, inwhich we have also renamed the relationship set to Manages2.

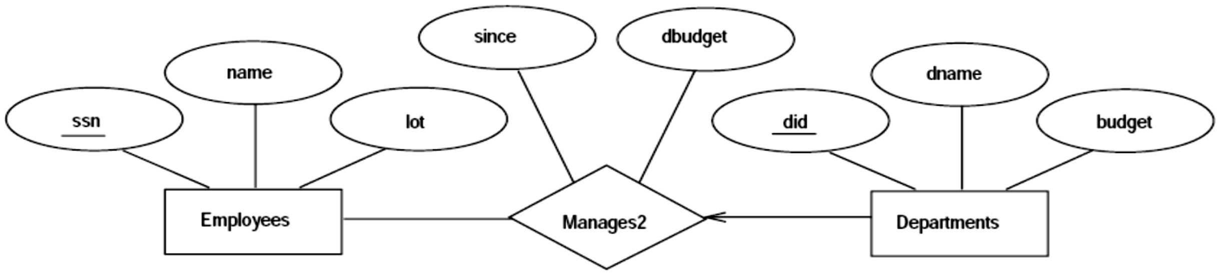


Figure 2.16 Entity versus Relationship

We can address these problems by associating dbudget with the appointment of the employee as manager of a group of departments. In this approach, we model the appointment as an entity set, sayMgrAppt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department.

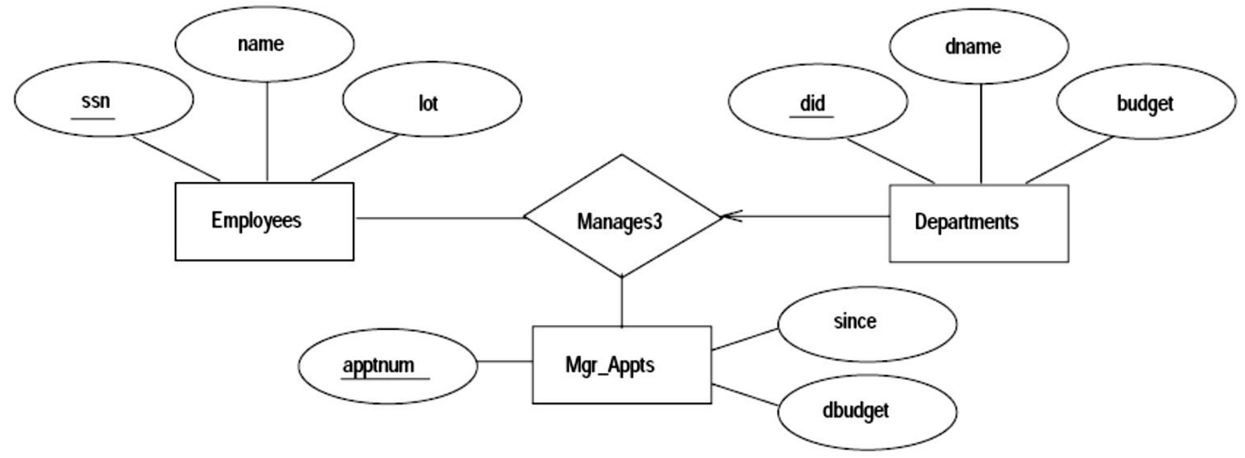


Figure 2.17 Entity Set versus Relationship

Binary versus Ternary Relationships *

Consider the ER diagram shown in Figure 2.18. It models a situation in which anemployee can own several policies, each policy can be owned by several employees, andeach dependent can be covered by several policies.Suppose that we have the following additional requirements:

1. A policy cannot be owned jointly by two or more employees.
2. Every policy must be owned by some employee.
3. Dependents is a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent).

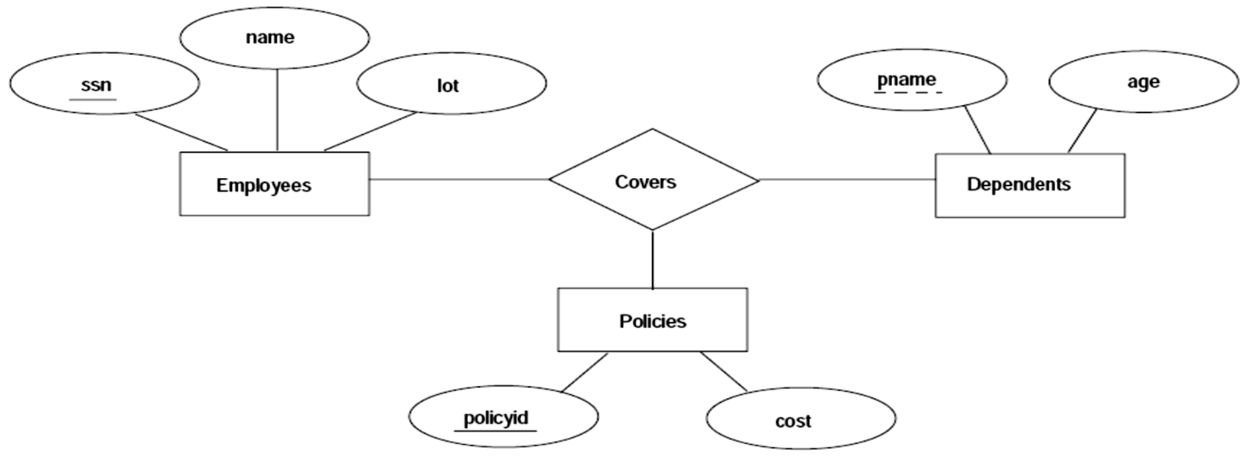


Figure 2.18 Policies as an Entity Set

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case). Even ignoring the third point above, the best way to model this situation is to use twobinary relationships, as shown in Figure 2.19.

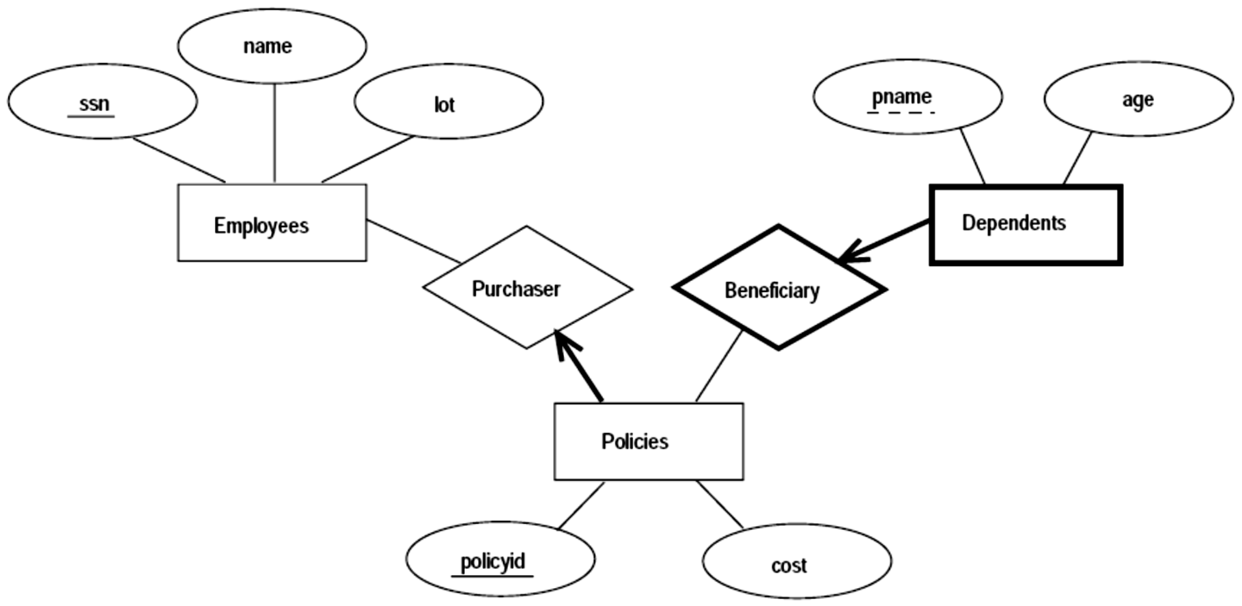


Figure 2.19 Policy Revisited

Aggregation versus Ternary Relationships *

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationshipset to an entity set (or second relationship set). The choice may also be guided bycertain integrity constraints that we want to express. For example, consider the ERdiagram shown in Figure 2.13.

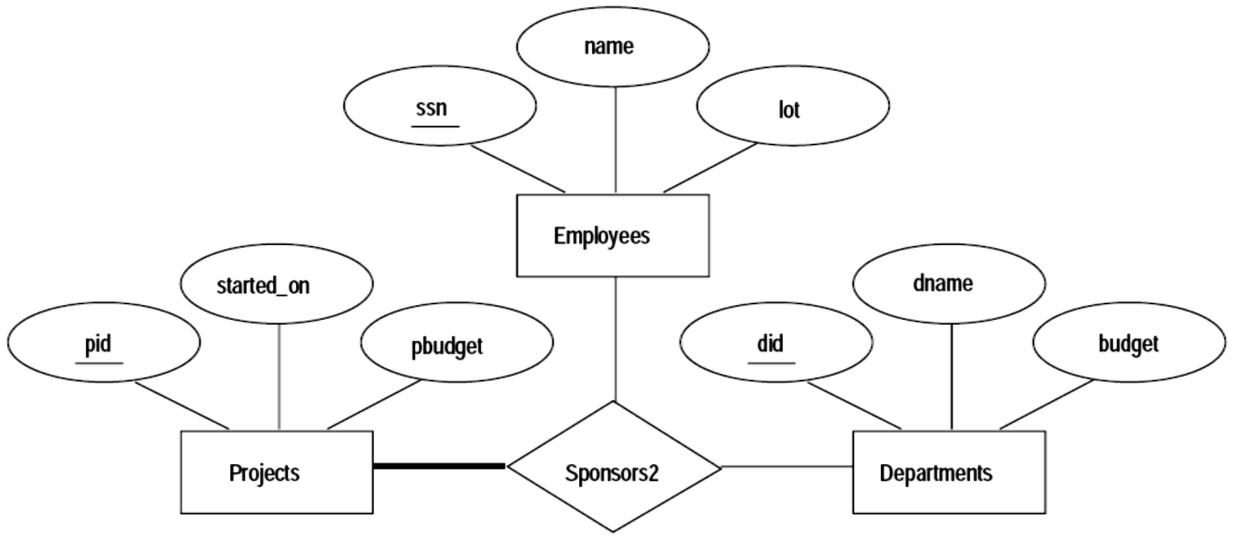


Figure 2.20 Using a Ternary Relationship instead of Aggregation

According to this diagram, a project can be sponsoredby any number of departments, a department can sponsor one or more projects, andeach sponsorship is monitored by one or more employees. If we don't need to recordthe until attribute of Monitors, then we might reasonably use a ternary relationship,say, Sponsors2.

CONCEPTUAL DESIGN FOR LARGE ENTERPRISES *

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model such as ER diagrams for conceptual design insuch an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, butit allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

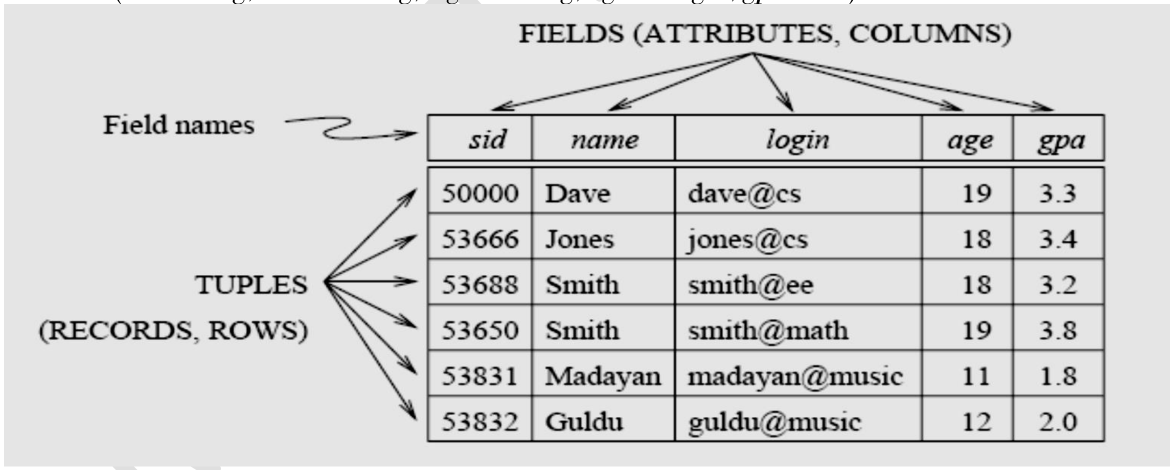
An alternative approach is to develop separate conceptual schemas for different user groups and to then integrate these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations schema integration cannot be avoided. for example,when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to heterogeneous data sources, often maintained by different organizations.

UNIT-II
INTRODUCTION TO RELATIONAL MODEL

UNIT II
Introduction to the Relational Model – **Integrity Constraint Over relations** – **Enforcing Integrity constraints** – Querying relational data – Logical database Design – Introduction to Views – Destroying /altering Tables and Views. **Relational Algebra – Selection and projection set operations – renaming – Joins – Division** – Examples of Algebra queries – **Relational calculus – Tuple relational Calculus – Domain relational calculus**.

Introduction to Relational Model:

Codd proposed the relational data model in 1970. At that time most database systemswere based on one of two older data models (the hierarchical model and the networkmodel); the relational model revolutionized the database eid and largely supplantedthese earlier models.The relational model is very simple and elegant; a database is a collection of one or morerelations, where each relation is a table with rows and columns. This simple tabularrepresentation enables even novice users to understand the contents of a database,and it permits the use of simple, high-level languages to query the data. The majoradvantages of the relational model over the older data models are its simple datarepresentation and the ease with which even complex queries can be expressed.
The main construct for representing data in the relational model is a **relation**. Arelation consists of a **relation schema** and a **relation instance**. The relation instance is a table, and the relation schema describes the column heads for the table. We firstdescribe the relation schema and then the relation instance. The schema specifies therelation's name, the name of each **field** (or **column**, or **attribute**), and the **domain**of each field. A domain is referred to in a relation schema by the **domain name** and
has a set of associated **values**.
We use the example of student information in a university database from Chapter 1to illustrate the parts of a relation schema:
Students(*sid:* string, *name:* string, *login:* string, *age:* integer, *gpa:* real)



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

This says, for instance, that the field named *sid*has a domain named string. The setof values associated with domain string is the set of all character strings.
An **instance** of a relation is a set of**tuples**, also called **records**, in which each tuple has the same number of fields as therelation schema. A relation instance can be thought of as a *table* in which each tupleis a *row*, and all rows have the same number of fields.
Arelation schema specifies the domain of each field or column in the relation instance.
These **domain constraints** in the schema specify an important condition that wewant each instance of the relation to satisfy: The values that appear in a column mustbe drawn from the domain associated

with that column. Thus, the domain of a fieldis essentially the *type* of that field, in programming language terms, and restricts thevalues that can appear in the field.
Domain constraints are so fundamental in the relational model that we will henceforthconsider only relation instances that satisfy them; therefore, *relation instance* means*relation instance that satisfies the domain constraints in the relation schema*.
The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality**of a relation instance is the number of tuples in it. The degree of therelation (the number of columns) is five, and the cardinality of this instance is six.
A **relational database** is a collection of relations with distinct relation names. The**relational database schema** is the collection of schemas for the relations in thedatabase. For example, in Chapter 1, we discussed a university database with relationscalled Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets In. An**instance** of a relational database is a collection of relation instances, one per relationschema in the database schema; of course, each relation instance must satisfy thedomain constraints in its schema.

CONSTRAINTS OVER RELATIONS

A database is only as good as the information stored in it, and a DBMS must there for help prevent the entry of incorrect information. An integrity constraint (IC) is a condition that is specified on a database schema, and restricts the data that can bestored in an instance of the database. If a database instance satisfies all the integrityconstraints specified on the database schema, it is a legal instance. A DBMS enforces integrity constraints, in that it permits only legal instances to be stored in the database.
Integrity constraints are specified and enforced at different times:
1. When the DBA or end user defines a database schema, he or she specifies the ICsthat must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallowschanges to the data that violate the specified ICs. (In some situations, rather thandisallow the change, the DBMS might instead make some compensating changesto the data to ensure that the database instance satisfies all ICs. In any case,changes to the database are not allowed to create an instance that violates any
IC.)

Key Constraints

Consider the Students relation and the constraint that no two students have the samestudent id. This IC is an example of a key constraint. A key constraint is a statementthat a certain minimal subset of the fields of a relation is a unique identifier for a tuple.A set of fields that uniquely identifies a tuple according to a key constraint is calleda candidate key for the relation; we often abbreviate this to just key. In the case ofthe Students relation, the (set of fields containing just the) sidfield is a candidate key
Let us take a closer look at the above definition of a (candidate) key. There are twoparts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, includingthe key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.
The set {sid, name} is an example of a superkey, which is a set of fields that contains a key.

Specifying Key Constraints in SQL-92

In SQL we can declare that a subset of the columns of a table constitute a key byusing the UNIQUE constraint. At most one of these `candidate' keys can be declaredto be a primary key, using the PRIMARY KEY constraint.

CREATE TABLE Students (sidCHAR(20),name CHAR(30),login CHAR(20),age INTEGER,gpa REAL,UNIQUE (name, age),CONSTRAINT StudentsKey PRIMARY KEY (sid))

Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored inanother relation. An IC involving both relations mustbe specified if a DBMS is to make such checks. The most common IC involving tworelations is a foreign key constraint.
Suppose that in addition to Students, we have a second relation:

Enrolled(sid: string, cid: string, grade: string)

The sidfield of Enrolled is called a foreignkey and refers to Students. The foreign key in the referencing relation (Enrolled, inour example) must match the primary key of the referenced relation (Students).

Specifying Foreign Key Constraints in SQL-92

Let us define Enrolled(sid: string, cid: string, grade: string):
CREATE TABLE Enrolled (sid CHAR(20),cid CHAR(20),grade CHAR(10),PRIMARY KEY (sid, cid),FOREIGN KEY (sid) REFERENCES Students)

General Constraints

Domain, primary key, and foreign key constraints are considered to be a fundamentalpart of the relational data model and are given special attention in most commercialsystems. Sometimes, however, it is necessary to specify more general constraints.
For example, we may require that student ages be within a certain range of values;given such an IC specification, the DBMS will reject inserts and updates that violatethe constraint. This is very useful in preventing data entry errors.Current relational database systems support such general constraints in the form oftable constraints and assertions. Table constraints are associated with a single tableand are checked whenever that table is modified. In contrast, assertions involve severaltables and are checked whenever any of these tables is modified. Both table constraintsand assertions can use the full power of SQL queries to specify the desired restriction.

ENFORCING INTEGRITY CONSTRAINTS

ICs are specified when a relation is created and enforced whena relation is modified. The impact of domain, PRIMARY KEY, and UNIQUE constraintsis straightforward: if an insert, delete, or update command causes a violation, it isrejected.
The following insertionviolates the primary key constraint because there is already a tuple with the sid 53688,and it will be rejected by the DBMS:
INSERTINTO Students (sid, name, login, age, gpa)VALUES (53688, `Mike', `mike@ee', 17, 3.4);
The following insertion violates the constraint that the primary key cannot containnull:
INSERTINTO Students (sid, name, login, age, gpa)VALUES (null, `Mike', `mike@ee', 17, 3.4);
The impact of foreign key constraints is more complex because SQL sometimes tries torectify a foreign key constraint violation instead of simply rejecting the change.
SQL-92 provides several alternative ways to handle foreign key violations. We mustconsider three basic questions:
1. What should we do if an Enrolled row is inserted, with a sid column value thatdoes not appear in any row of the Students table?
In this case the INSERT command is simply rejected.
2. What should we do if a Students row is deleted?
The options are:

- Delete all Enrolled rows that refer to the deleted Students row.

- Disallow the deletion of the Students row if an Enrolled row refers to it.
- Set the sid column to the sid of some (existing) `default' student, for everyEnrolled row that refers to the deleted Students row.
- For every Enrolled row that refers to it, set the sid column to null. In ourexample, this option conflicts with the fact that sid is part of the primarykey of Enrolled and therefore cannot be set to null. Thus, we are limited tothe first three options in our example, although this fourth option (settingthe foreign key to null) is available in the general case.

3. What should we do if the primary key value of a Students row is updated?
The options here are similar to the previous case.
SQL-92 allows us to choose any of the four options on DELETE and UPDATE. For example,
we can specify that when a Students row is deleted, all Enrolled rows that refer toit are to be deleted as well, but that when the sid column of a Students row is modified,this update is to be rejected.
CREATE TABLE Enrolled (sid CHAR(20),cid CHAR(20),grade CHAR(10),PRIMARY KEY (sid, cid),FOREIGN KEY (sid) REFERENCES StudentsON DELETE CASCADEON UPDATE NO ACTION)
The options are specified as part of the foreign key declaration. The default option isNO ACTION, which means that the action (DELETE or UPDATE) is to be rejected.

QUERYING RELATIONAL DATA

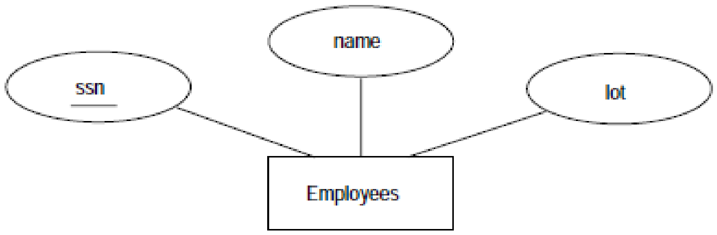
A relational database query (query, for short) is a question about the data, and theanswer consists of a new relation containing the result. A querylanguage is a specialized language for writing queries. SQL is the most popular commercial query language for a relational DBMS.
We can retrieve rowscorresponding to students who are younger than 18 with the following SQL query:
SELECT *FROM Students SWHERE S.age< 18
The symbol * means that we retain all fields of selected tuples in the result. Tounderstand this query, think of S as a variable that takes on the value of each tuplein Students, one tuple after the other. The condition S.age< 18 in the WHERE clausespecifies that we want to select only tuples in which the age field has a value less than18.
We can also combine information in the Students and Enrolled relations. If we want toobtain the names of all students who obtained an A and the id of the course in whichthey got an A, we could write the following query:
SELECT S.name, E.cidFROM Students S, Enrolled EWHERE S.sid = E.sid AND E.grade = `A'
This query can be understood as follows: If there is a Students tuple S and an Enrolledtuple E such that S.sid = E.sid (so that S describes the student who is enrolled in E)and E.grade = `A', then print the student's name and the course id."

LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design.Given an ER diagram describing a database, there is a standard approach to generatinga relational database schema that closely approximates the ER design.We now describe how to translate an ER diagram into a collection of tableswith associated constraints, i.e., a relational database schema.

1. Entity Sets to Tables

An entity set is mapped to a relation in a straightforward way: Each attribute of theentity set becomes an attribute of the table. Note that we know both the domain ofeach attribute and the (primary) key of an entity set.
Consider the Employees entity set with attributes ssn, name, and lot shown in Figure3.8. A possible instance of the Employees entity set, containing three EmployeesFigure 3.8



The Employees Entity Set entities, is shown in Figure 3.9 in a tabular format.

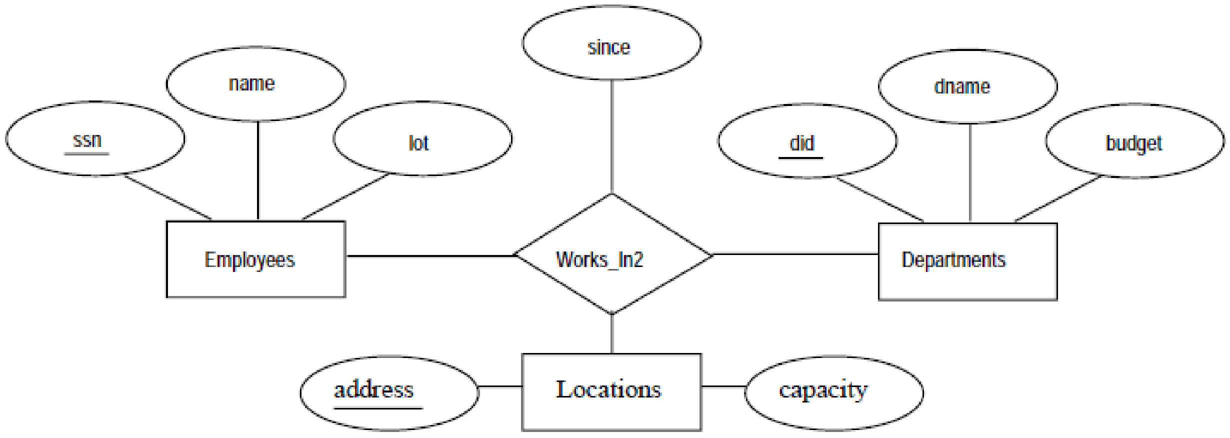
ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

2. Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model.We begin by considering relationship sets without key and participation constraints. To representa relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship. Thus, the attributes of the relationinclude:

- The primary key attributes of each participating entity set, as foreign key fields.
- The descriptive attributes of the relationship set.

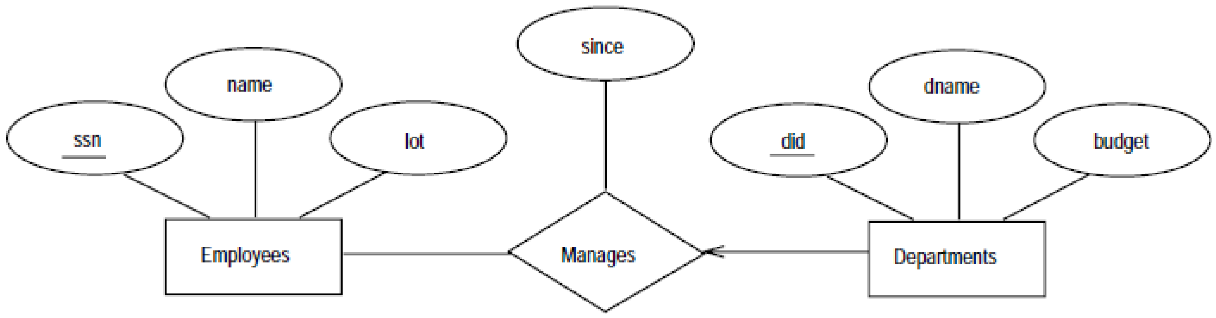
The set of non-descriptive attributes is a superkey for the relation. If there are no keyconstraints, this set of attributes is a candidate key.
Consider the Works In2 relationship set shown in Figure 3.10. Each department haso_ces in several locations and we want to record the locations at which each employeeworks.



All the available information about the Works In2 table is captured by the followingSQL definition: CREATE TABLE Works In2 (ssn CHAR(11),did INTEGER,address CHAR(20),since DATE, PRIMARY KEY (ssn, did, address),FOREIGN KEY (ssn) REFERENCES Employees,FOREIGN KEY (address) REFERENCES Locations,FOREIGN KEY (did) REFERENCES Departments)

3. Translating Relationship Sets with Key Constraints

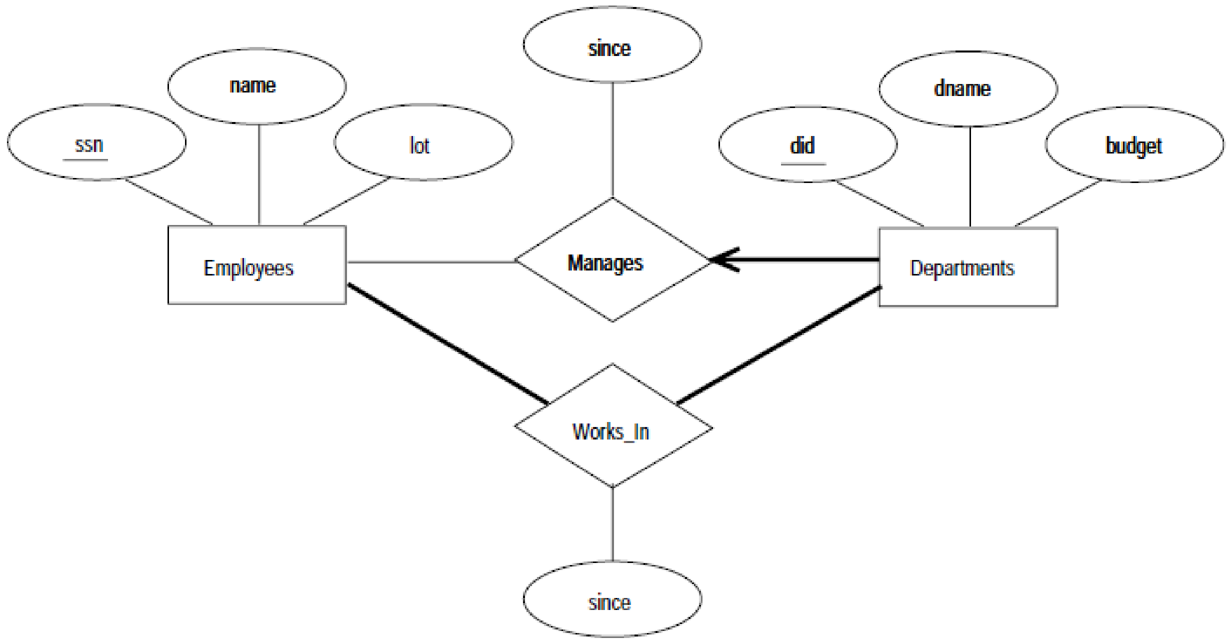
If a relationship set involves n entity sets and some m of them are linked via arrowsin the ER diagram, the key for any one of these m entity sets constitutes a key forthe relation to which the relationship set is mapped. Thus we have m candidate keys,and one of these should be designated as the primary key.Consider the relationship set Manages shown in Figure 3.12.



However, because each department hasat most one manager, no two tuples can have the same did value but differ on the ssnvalue. A consequence of this observation is that did is itself a key for Manages;
The Manages relation can bedefined using the following SQL statement:
CREATE TABLE Manages (ssnCHAR(11),did INTEGER,since DATE,PRIMARY KEY (did), FOREIGN KEY (ssn) REFERENCES Employees,FOREIGN KEY (did) REFERENCES Departments)
A second approach to translating a relationship set with key constraints is often superiorbecause it avoids creating a distinct table for the relationship set. The ideais to include the information about the relationship set in the table corresponding tothe entity set with the key, taking advantage of the key constraint.
This approach eliminates the need for a separate Manages relation, and queries askingfor a department's manager can be answered without combining information from tworelations. The only drawback to this approach is that space could be wasted if severaldepartments have no managers.

4. Translating Relationship Sets with Participation Constraints

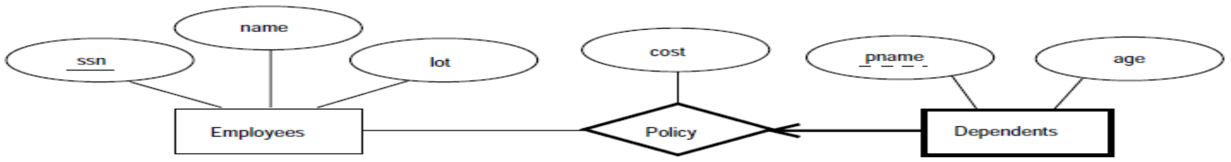
Consider the ER diagram in Figure 3.13, which shows two relationship sets, Managesand Works In.



Every department is required to have a manager, due to the participation constraint,and at most one manager, due to the key constraint.
The following SQL statementreflects the second translation approach discussed in Section 3.5.3, and uses the keyconstraint:
CREATE TABLE Dept Mgr (did INTEGER,dname CHAR(20),budget REAL,ssn CHAR(11) NOT NULL,since DATE,PRIMARY KEY (did),FOREIGN KEY (ssn) REFERENCES EmployeesON DELETE NO ACTION)

5. Translating Weak Entity Sets

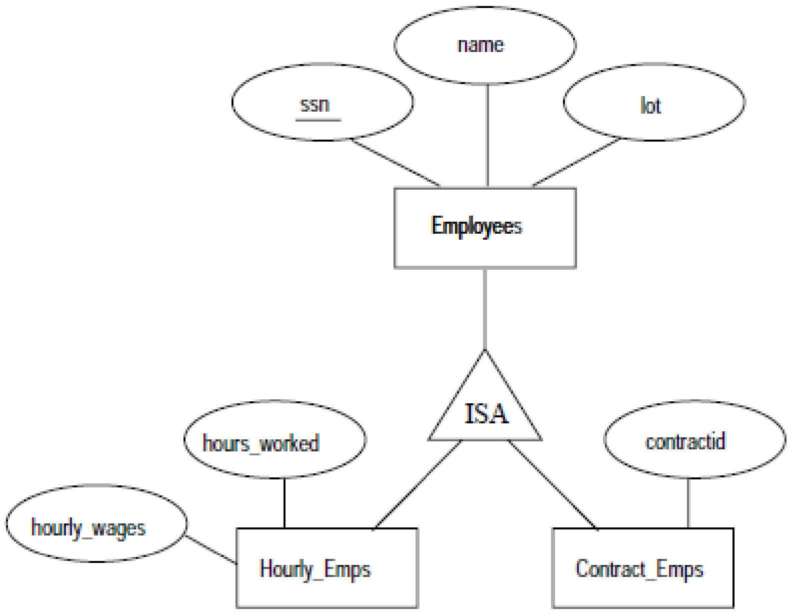
A weak entity set always participates in a one-to-many binary relationship and has akey constraint and total participation.Consider the Dependents weak entity set shown in Figure 3.14, with partial key pname.



A Dependents entity can be identified uniquely only if we take the key of the owningEmployees entity and the pname of the Dependents entity, and the Dependents entitymust be deleted if the owning Employees entity is deleted.
We can capture the desired semantics with the following de_nition of the Dep Policyrelation:
CREATE TABLE Dep Policy (pname CHAR(20),age INTEGER,cost REAL,ssn CHAR(11), PRIMARY KEY (pname, ssn),FOREIGN KEY (ssn) REFERENCES EmployeesON DELETE CASCADE)
Observe that the primary key is hpname, ssni, since Dependents is a weak entity.

6. Translating Class Hierarchies

We present the two basic approaches to handling ISA hierarchies by applying them tothe ER diagram shown in Figure 3.15:

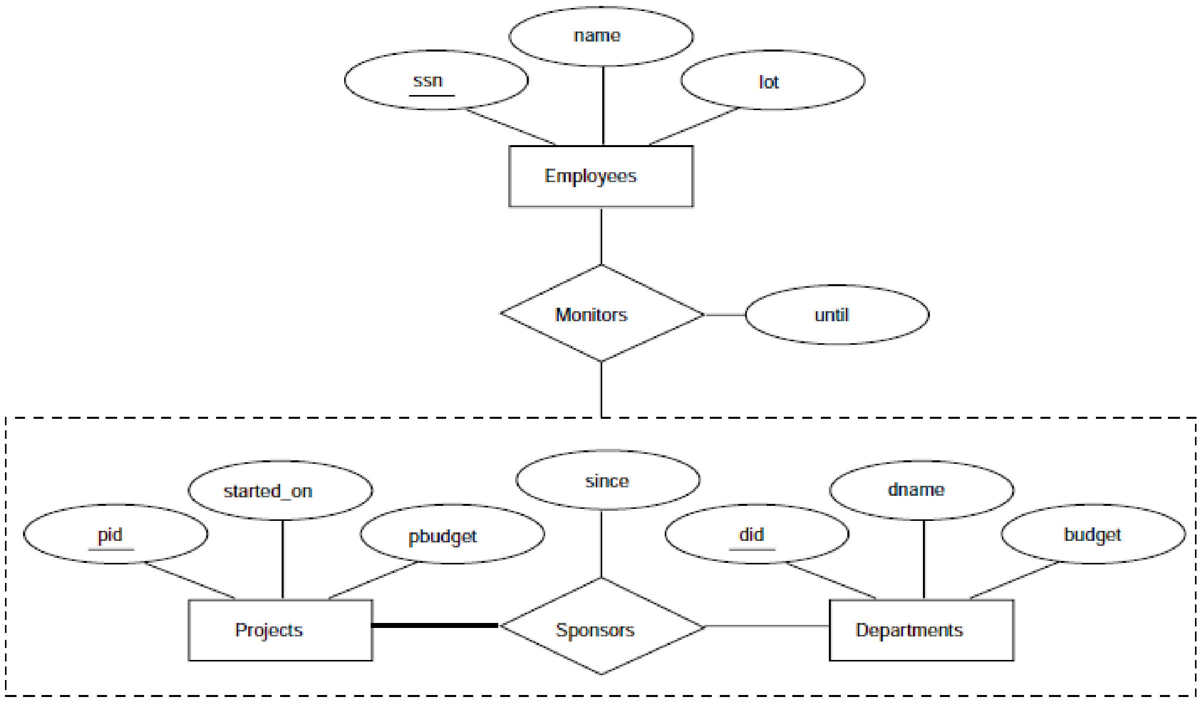


1. We can map each of the entity sets Employees, Hourly Emps, and Contract Empsto a distinct relation. The Employees relation is created as in Section 2.2. Wediscuss Hourly Emps here; Contract Emps is handled similarly. The relation forHourly Emps includes the hourly wages and hours worked attributes of Hourly Emps.

2. Alternatively, we can create just two relations, corresponding to Hourly Empsand Contract Emps. The relation for Hourly Emps includes all the attributesof Hourly Emps as well as all the attributes of Employees (i.e., ssn, name, lot,hourly wages, hours worked).
The first approach is general and is always applicable. Queries in which we want toexamine all employees and do not care about the attributes specific to the subclassesare handled easily using the Employees relation.
The second approach is not applicable if we have employees who are neither hourlyemployees nor contract employees, since there is no way to store such employees. Also,if an employee is both an Hourly Emps and a Contract Emps entity, then the nameand lot values are stored twice.

7. Translating ER Diagrams with Aggregation

Translating aggregation into the relational model is easy because there is no real distinctionbetween entities and relationships in the relational model.



Consider the ER diagram shown in Figure 3.16. The Employees, Projects, and Departmentsentity sets and the Sponsors relationship set are mapped as described inprevious sections. For the Monitors relationship set, we create a relation with thefollowing attributes: the key attributes of Employees (ssn), the key attributes of Sponsors (did, pid), and the descriptive attributes of Monitors (until). This translation isessentially the standard mapping for a relationship set, as described in Section 3.5.2.
There is a special case in which this translation can be re_ned further by droppingthe Sponsors relation. Consider the Sponsors relation. It has attributes pid, did, andsince, and in general we need it (in addition to Monitors) for two reasons:
1. We have to record the descriptive attributes (in our example, since) of the Sponsorsrelationship.

2. Not every sponsorship has a monitor, and thus some hpid, didipairs in the Sponsors relation may not appear in the Monitors relation.
However, if Sponsors has no descriptive attributes and has total participation in Monitors, every possible instance of the Sponsors relation can be obtained by looking atthe hpid, didi columns of the Monitors relation. Thus, we need not store the Sponsorsrelation in this case.

INTRODUCTION TO VIEWS

A view is tables whose row is not explicitly stored in the database but is computedas needed from a view definition. Consider the Students and Enrolled relations.Suppose that we are often interested in finding the names and student identifiers ofstudents who got a grade of B in some course, together with the cid for the course.
We can de_ne a view for this purpose. Using SQL-92 notation:

CREATE VIEW B-Students (name, sid, course)AS SELECT S.sname, S.sid, E.cidFROM Students S, Enrolled EWHERE S.sid = E.sid AND E.grade = `B';
The view B-Students has three fields called name, sid, and course with the samedomains as the _eldssname and sid in Students and cid in Enrolled.
This view can be used just like a base table, or explicitly stored table, in defining newqueries or views.

Updates on Views

The motivation behind the view mechanism is to tailor how users see the data. Usersshould not have to worry about the view versus base table distinction. This goal isindeed achieved in the case of queries on views; a view can be used just like any otherrelation in de_ning a query.
The SQL-92 standard allows updates to be specified only on views that are definedon a single base table using just selection and projection, with no use of aggregateoperations. Such views are called updatable views.
An important observation is that an INSERT or UPDATE may change the underlyingbase table so that the resulting (i.e., inserted or modified) row is not in the view!

Need to Restrict View Updates

While the SQL-92 rules on updatable views are more stringent than necessary, thereare some fundamental problems with updates specified on views, and there is goodreason to limit the class of views that can be updated.

DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., deleteall the rows and remove the table definition information), we can use the DROP TABLEcommand. For example, DROP TABLE Students RESTRICT destroys the Students tableunless some view or integrity constraint refers to Students; if so, the command fails.
If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any referencingviews or integrity constraints are (recursively) dropped as well; one of thesetwo keywords must always be speci_ed. A view can be dropped using the DROP VIEWcommand, which is just like DROP TABLE.
ALTER TABLE modi_es the structure of an existing table.ALTER TABLE can also be used to deletecolumns and to add or drop integrity constraints on a table; we will not discuss theseaspects of the command beyond remarking that dropping columns is treated verysimilarly to dropping tables or views.Query languages are specialized languages for asking questions, or queries, that involve the data in a database.

RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relationalmodel. Queries in algebra are composed using a collection of operators. Afundamental property is that every operator in the algebra accepts (one or two) relationinstances as arguments and returns a relation instance as the result. This propertymakes it easy to compose operators to form a complex query|a

relational algebraexpression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. Wedescribe the basic operators of the algebra (selection, projection, union, cross-product,and difference), as well as some additional operators that can be defined in terms ofthe basic operators.
Each relational query describes a step-by-step procedure for computing the desiredanswer, based on the order in which operators are applied in the query. The proceduralnature of the algebra allows us to think of an algebra expression as a recipe, or aplan, for evaluating a query, and relational systems in fact use algebra expressions torepresent query evaluation plans.

1. Selection and Projection

Relational algebra includes operators to select rows from a relation (σ) and to projectcolumns (π). These operations allow us to manipulate data in a single relation.
We canretrieve rows corresponding to expert sailors by using the operatorσ. The expressionrating>8(S2)evaluates to the relation.The subscript rating>8 specifies theselection criterion to be applied while retrieving tuples.

The selection operator specifies the tuples to retain through a selection condition.In general, the selection condition is a boolean combination (i.e., an expression usingthe logical connectives ^ and v) of terms that have the form attribute op constant orattribute1 op attribute2, where op is one of the comparison operators {<, <=, =,!=, >=} or >.

2. Projection

The projection operator π allows us to extract columns from a relation; for example,we can find out all sailor names and ratings by using π. The expression πsname, rating(S2). The subscript sname, rating specifies the fields to be retained; the other fields are `projected out.' The schema of the result ofa projection is determined by the fields that are projected in the obvious way.Suppose that we wanted to find out only the ages of sailors. The expressionage(S2)evaluates to the relation shown in Figure 4.6. The important point to note is thatalthough three sailors are aged 35, a single tuple with age=35.0 appears in the resultof the projection.

3. Set Operations

The following standard operations on sets are also available in relational algebra: union(U), intersection (∩), set-difference (−), and cross-product (×).
Union: RUS returns a relation instance containing all tuples that occur in eitherrelation instance R or relation instance S (or both). R and S must be unioncompatible,and the schema of the result is defined to be identical to the schemaof R.
Two relation instances are said to be union-compatible if the following conditionshold:
-They have the same number of the fields,and
-Correspondingfields, taken in order from left to right, have the same domains.
Intersection: R∩S returns a relation instance containing all tuples that occur inboth R and S. The relations R and S must be union-compatible, and the schemaof the result is defined to be identical to the schema of R.

Set-difference: R − S returns a relation instance containing all tuples that occurin R but not in S. The relations R and S must be union-compatible, and theschema of the result is defined to be identical to the schema of R.
Cross-product: R×S returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S(in the same order as they appear in S). The result of R × S contains one tuple<r; s> (the concatenation of tuples r and s) for each pair of tuples r€R; s € S.The cross-product operation is sometimes called**Cartesian product.**

4. Renaming

We have been careful to adopt field name conventions that ensure that the result ofa relational algebra expression inherits field names from its argument (input) relationinstances in a natural way whenever possible.We introduce a renaming operatorρ for this purpose. The expression ρ(R(F),E)takes an arbitrary relational algebra expression E and returns an instance of a (new)relation called R. R contains the same tuples as the result of E, and has the sameschema as E, but some fields are renamed. The field names in relation R are thesame as in E, except for fields renamed in the renaming list F, which is a list ofterms having the form oldname → newname or position → newname. For ρ to bewell-defined, references to fields (in the form of oldnames or positions in the renaminglist) may be unambiguous, and no two fields in the result must have the same name.Sometimes we only want to rename fields or to (re)name the relation; we will thereforetreat both R and F as optional in the use of ρ. (Of course, it is meaningless to omitboth.)
For example, the expressionρ(C(1 → sid1, 5 → sid2), S1×R1) returns a relationthat contains the tuples shown in Figure 4.11 and has the following schema: C(sid1:integer, sname: string, rating: integer, age: real, sid2: integer, bid: integer,day: dates).

5. Joins

The join operation is one of the most useful operations in relational algebra and isthe most commonly used way to combine information from two or more relations.Although a join can be de_ned as a cross-product followed by selections and projections,joins arise much more frequently in practice than plain cross-products.
There are several variants ofthe join operation.

Condition Joins

The most general version of the join operation accepts a join condition c and a pair ofrelation instances as arguments, and returns a relation instance. The join condition isidentical to a selection condition in form. The operation is defined as follows:

$$R \times_c S = \sigma(R \times S)$$

Equijoin

A common special case of the join operation R × S is when the join condition consistssolely of equalities (connected by ^) of the form R.name1 = S.name2, that is,equalities between two fields in R and S. In this case, obviously, there is some redundancyin retaining both attributes in the result.
The join operation with this refinement is called equijoin.

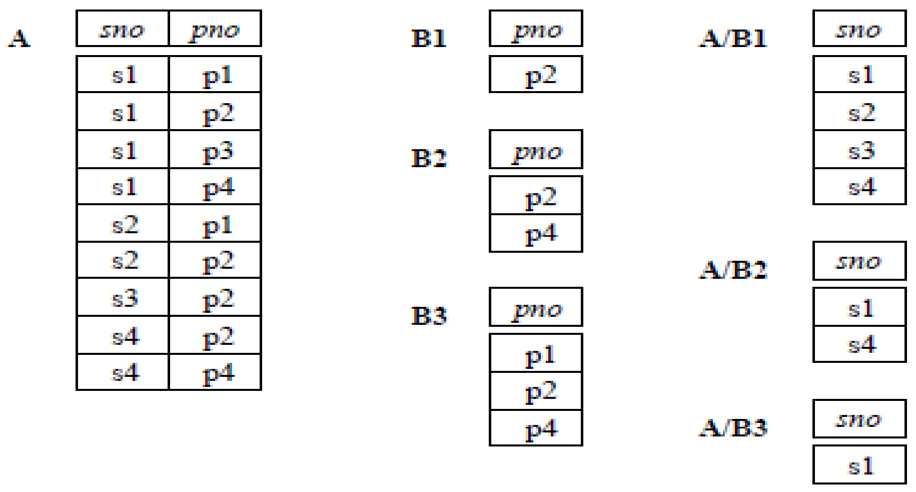
Natural Join

A further special case of the join operation R × S is an equijoin in which equalitiesare specified on allfields having the same name in R and S. In this case, we cansimply omit the join condition; the default is that the join condition is a collection ofequalities on all common fields. We call this special case a natural join, and it has thenice property that the result is guaranteed not to have two fields with the same name.

6. Division

Consider two relation instances A and B inwhich A has (exactly) two fields x and y and B has just one field y, with the samedomain as in A. We define the division operation A=B as the set of all x values (inthe form of unary tuples) such that for every y value in (a tuple of) B, there is a tuple<x,y> in A.

Eg:



A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Examples of Relational Algebra Queries

(Q1) Find the names of sailors who have reserved boat 103.

$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$

(Q2) Find the names of sailors who have reserved a red boat.

$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$

(Q3) Find the colors of boats reserved by Lubber.

$\pi_{color}((\sigma_{sname='Lubber'} Sailors) \bowtie Reserves \bowtie Boats)$

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra,which is procedural, the calculus is nonprocedural, or declarative, in that it allowsus to describe the set of answers without being explicit about how they should becomputed. Relational calculus has had a big influence on the design of commercialquery languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus that we present in detail is called the tuple relationalcalculus (TRC). Variables in TRC take on tuples as values. In another variant,calledthe domain relationalcalculus (DRC), the variables range over field values. TRC hashad more of an influence on SQL, while DRC has strongly influenced QBE.

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema asvalues. That is, every value assigned to a given tuple variable has the same numberand type of fields. A tuple relational calculus query has the form {T /p(T) }, whereT is a tuple variable and p(T) denotes a formula that describes T. we will shortlydefine formulas and queries rigorously. The result of this query is the set of all tuplest for which the formula p(T) evaluates to true with T = t.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. LetRel be a relation name, R and S be tuple variables, 'a'is an attribute of R, and 'b' anattribute of S. Let op denote an operator in the set {<,>,=,!=,<=,>=}.

An atomicformula is one of the following:

- R €Rel
- R.a op S.b
- R.a op constant, or constant op R.a

A formula is recursively defined to be one of the following, where p and q are themselvesformulas, and p(R) denotes a formula in which the variable R appears:

1. any atomic formula
2. ¬p, p ^ q, p **v**q, or p→q
3. R(p(R)), where R is a tuple variable
4. R(p(R)), where R is a tuple variable

In the last two clauses above, the quantifiers and are said to bind the variableR.A variable is said to be free in a formula or subformula (a formula contained in a larger formula) if the (sub)formula does not contain an occurrence of a quantifier thatbinds it.

Semantics of TRC Queries

Let each free variable in aformula F be bound to a tuple value. For the given assignment of tuples to variables,with respect to the given database instance, F evaluates to (or simply `is') true if oneof the following holds:

- F is an atomic formula R €Rel, and R is assigned a tuple in the instance ofrelation Rel.
- F is a comparison R.a op S.b, R.a op constant, or constant op R.a, and the tuplesassigned to R and S havefield values R.a and S.b that make the comparison true.
- F is of the form ¬p, and p is not true; or of the form p ^ q, and both p and q aretrue; or of the form p **v**q, and one of them is true, or of the form p → q and q istrue whenever p is true.
- F is of the form R(p(R)), and there is some assignment of tuples to the freevariables in p(R), including the variable R, that makes the formula p(R) true.
- F is of the form R(p(R)), and there is some assignment of tuples to the freevariables in p(R) that makes the formula p(R) true no matter what tuple isassigned to R.

Examples of TRC Queries

(Q1) Find the names and ages of sailors with a rating above 7.

{P /(S €Sailors(S.rating> 7 ^ P.name = S.sname^P.age = S.age)}

(Q2) Find the sailor name, boat id, and reservation date for each reservation.

{P /∃R €Reserves ∃S €Sailors(R.sid= S.sid^P.bid= R.bid^ P.day= R.day^ P.sname= S.sname)}

(Q3) Find the names of sailors who have reserved boat 103.

{P / {S €Sailors ∃R €Reserves(R.sid = S.sid^ R.bid = 103 ^ *P.sname= S.sname*)}

(Q4) Find the names of sailors who have reserved a red boat.

{P /∃S €Sailors ∃R €Reserves(R.sid= S.sid^P.sname= S.sname^ ∃B €Boats(B.bid= R.bid^B.color='red'))}

Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form {<x1, x2,....xn>/ p(<x1, x2.....xn>)}, where each x_i is either a domain variable or a constant and p(<x1, x2,.....xn>) denotes a DRC formula whose only free variables are the variables among the xi; 1 ≤ i ≤ n. The result of this query is the set of all tuples <x1, x2 . .....xn> for which the formula evaluates to true.

A DRC formula is defined in a manner that is very similar to the definition of a TRCformula. The main di_erence is that the variables are now domain variables. Let opdenote an operator in the set {<,>,=,!=, <=,>=} and let X and Y be domain variables.
An atomic formula in DRC is one of the following:

- <x1, x2, ...xn> € Rel, where Rel is a relation with n attributes; each xi, 1 ≤ i ≤ n is either a variable or a constant.
- X op Y
- X op constant , or constant op X

A **formula** is recursively de_ned to be one of the following, where p and q are themselves formulas, and **p(X)** denotes a formula in which the variable X appears:

- any atomic formula
- ¬p, p ^ q, p **v** q, or p → q
- ∃X(p(X)), where X is a domain variable
- ∀X(p(X)), where X is a domain variable

Examples of DRC Queries

(Q1) Find all sailors with a rating above 7.
{<I,N, T,A> / <I,N, T,A>€Sailors ^ T >7}

(Q2) Find the names of sailors who have reserved boat 103.
{ <N> /∃I, T ,A(<I ,N ,T ,A>€ Sailors ^ ∃Ir,Br,D(<Ir,Br,D>€ Reserves ^Ir = I ^ Br = 103))}

(Q3) Find the names of sailors who have reserved a red boat.
{<N> /∃I, T,A(<I,N, T,A>€Sailors^ ∃<I,Br,D>€Reserves ^∃<B, BN, 'red'>€Boats)}

EXPRESSIVE POWER OF ALGEBRA AND CALCULUS

We have discussed two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus.

Consider the query {S /¬(S € Sailors)g. This query is syntactically correct. However,it asks for all tuples S such that S is not in (the given instance of) Sailors. The set ofsuch S tuples is obviously infinite, in the context of infinite domains such as the set ofall integers. This simple example illustrates an unsafe query. It is desirable to restrictrelational calculus to disallow unsafe queries.
Consider a set I ofrelation instances, with one instance per relation that appears in the query Q. Let Dom(Q; I) be the set of all constants that appear in these relation instances I or inthe formulation of the query Q itself. Since we only allow finite instances I, Dom(Q; I)is also finite.

For a calculus formula Q to be considered safe, at a minimum we want to ensure thatfor any given I, the set of answers for Q contains only values that are in Dom(Q; I).
We therefore define a safe TRC formula Q to be a formula such that:
1. For any given I, the set of answers for Q contains only values that are in Dom(Q; I).
2. For each subexpression of the form ∃R(p(R)) in Q, if a tuple r (assigned to variableR) makes the formula true, then r contains only constants in Dom(Q; I).
3. For each subexpression of the form ∀R(p(R)) in Q, if a tuple r (assigned to variableR) contains a constant that is not in Dom(Q; I), then r must make the formulatrue.

The query Q = {S /¬(S € Sailors)} is unsafe by this definition. Dom(Q,I) is theset of all values that appear in (an instance I of) Sailors. The answer to this query obviously includes values that do notappear in Dom(Q; S1).

In the view of expressiveness, we can show that every query that can beexpressed using a safe relational calculus query can also be expressed as a relationalalgebra query. The expressive power of relational algebra is often used as a metric ofhow powerful a relational database query language is. If a query language can expressall the queries that we can express in relational algebra, it is said to be relationallycomplete. A practical query language is expected to be relationally complete; in addition,commercial query languages typically support features that allow us to expresssome queries that cannot be expressed in relational algebra.

UNIT-III
INTRODUCTION TO SQL

The Form of a Basic SQL Query – Examples of Basic SQL Queries – Introduction to Nested Queries – Correlated Nested Queries, Set – Comparison Operators – Aggregate Operators – NULL values – Comparison using Null values – Logical connectivites – AND, OR and NOT – Impact on SQL Constructs – Outer Joins – Disallowing NULL values – Complex Integrity Constraints in SQL, Triggers and Active Data bases.

1. Introduction to SQL:

Structured Query Language (SQL) is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and System-R projects (1974 - 1977). Codd introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area. The SQL-92 is standardized with ANSI/ISO standard for SQL, which is called SQL-99.
The SQL language has several aspects to it:

- **The Data Definition Language (DDL):** The set of SQL commands like creation, deletion, and modification of definitions for tables and views. Integrity constraints can be defined on tables, either when the table is created or later. The DDL also provides commands for specifying access rights or privileges to tables and views.
- **The Data Manipulation Language (DML):** This subset of SQL allows users to pose queries and to insert, delete, and modify rows.
- **Triggers and Advanced Constraints:** The new SQL:1999 standard includes support for triggers, which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.
- **Security:** SQL provides mechanisms to control user's access to data objects such as tables and views.
- **Transaction management:** Various commands allow a user to explicitly control aspects of how a transaction is to be executed.
- **Client-server execution and remote database access:** These commands control how a client application program can connect to an SQL database server, or access data from a database over a network.
- **Advanced Features:** Integrity Constraints and Assertions.

2. THE FORM OF A BASIC SQL QUERY

Let we define the syntax of a simple SQL query and explains its meaning through a conceptual evaluation strategy. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

SELECT [DISTINCT] **select-list**
FROM **from-list**
WHERE **qualification**

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.
Let us consider a simple query:
(Q 1) Find the names and ages of all sailors.
SQL> **SELECT DISTINCT S.sname, S.age FROM Sailors S;**

Consider the following instances from sailors database includes sailors, Boats and Reserves tables:

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance 53 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance *R2* of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance *B1* of Boats

These instances of sailors, boats and reserves relations are used in writing queries.
(Q 2) Find all sailors with a rating above 7.

```
SQL> SELECT S.sid, S.sname, S.rating, S.age
         FROM Sailors AS S
         WHERE S.rating > 7;
```

Let we now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression

op expression, where op is one of the comparison operators {<=, =, <>, >=, >}.2 An expression is a column name, a constant, or an (arithmetic or string) expression.

- The **DISTINCT** keyword is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated.

In brief, the conceptual evaluation strategy performed in the following manner:

1. Compute the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

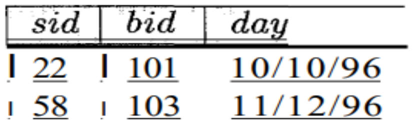
Describe the evaluation strategy with an example.

(Q 3) Find the names of sailors who have reserved boat number 103.

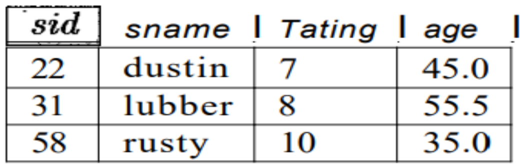
It can be expressed in SQL as follows.

SQL> SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103;

The evaluation of the above query performed by cross product of instances S4 and R3.



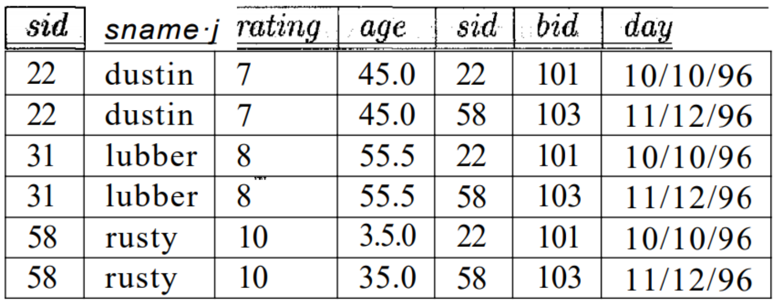
sid	bid	day
22	101	10/10/96
58	103	11/12/96



sid	sname	Tating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Instance of Reserves Relation R3 **Instance of Sailors Relation S4**

The cross product of these instances of R3 and S4 performed in the following manner.



sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	3.5.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure: S4 X R3

The second step is to apply the qualification S.sid = R.sid AND R.bid=103. The third step is to eliminate unwanted columns; only **sname** appears in the SELECT clause. Finally, displays unique values in the result.

sname
Rusty

2.1. Examples of Basic SQL Queries:

(Q 4) Find the sid's of sailors who have reserved a Red boat.

```
SQL>  SELECT R.sid
      FROM Boats B, Reserves R
      WHERE B.bid = R.bid and B.color = 'red';
```

(Q 5) Find the colors of boats reserved by Lubber.

```
SQL>  SELECT B.color
      FROM Sailors S, Reserves R, Boats B
      WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber';
```

(Q 6) Find the names of sailors who have reserved at least one boat.

```
SQL>  SELECT S.sname
      FROM Sailors S, Reserves R
      WHERE S.sid = R.sid;
```

2.2. Expressions and Strings in the SELECT Command:

SQL supports a more general version of the select-list than just a list of colu1nn8. Each item in a select-list can be of the form expression AS column name, where expression is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants, and column name is a new name for this column in the output of the query.

(Q 7) Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
SQL> SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2
      WHERE S.sid = R1.sid and S.sid = R2.sid and R1.day = R2.day and R1.bid <>
      R2.bid;
```

(Q 8) Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SQL> SELECT S.age FROM Sailors S WHERE S.sname LIKE 'B_%B';
```

3. UNION, INTERSECT, AND EXCEPT :

SQL provides three set-manipulation constructs that extend the basic query. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names **UNION, INTERSECT, and EXCEPT**. SQL also provides other set operations: **IN** (to check if an element is in a given set), **op ANY, op ALL** (to compare a value with the elements in a given set, using comparison operator op), and **EXISTS** (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning.

(Q 9) Find the names of sailors who have reserved a red or a green boat.

SQL> SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND (B.color = 'red' OR B.color = 'green');

(OR)

SQL> SELECT S.snarne FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

UNION

SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

(Q 10) Find the names of sailor's who have reserved both a red and a green boat?

SQL> SELECT FROM WHERE S.sname Sailors S, Reserves RI, Boats BI, Reserves R2, Boats B2 S.sid = Rl.sid AND R1.bid = Bl.bid AND S.sid = R2.sid AND R2.bid = B2.bid AND B1.color='red' AND B2.color = 'green';

(OR)

SQL> SELECT S.snarne FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

INTERSECT

SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

(Q 11) Find the sid's of all sailors who have reserved red boats but not green boats?

SQL> SELECT S.sid FROM Sailors S, Reserves R, Boats WHERE B S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

EXCEPT

SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

4. NESTED QUERIES:

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a sub query. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a sub query and appears as part of the main query. A sub query typically appears within the WHERE clause of a query. Sub queries can sometimes appear in the FROM clause or the HAVING clause.

4.1 Introduction to Nested Queries:

As an example, let us rewrite the following query, which we discussed earlier, using a nested sub query:

(Q l2) Find the names of sailors who have reserved boat 103.

SQL> SELECT S.sname FROM Sailors WHERE S S.sid IN (SELECT R.sid FROM Reserves R WHERE R.bid = 103);

The best way to understand a nested query is to think of it in terms of a conceptual evaluation strategy. In our example, the strategy consists of examining rows in Sailors and, for each such row, evaluating the sub query over Reserves. In general, the conceptual evaluation strategy that we presented for defining the semantics of a query can be extended to cover nested queries as follows: Construct the cross-product of the tables in the FROM clause of the top-level query as before. For each row in the cross-product, while testing the qualification in the WHERE clause, (re)compute the sub query. Of course, the sub query might itself contain another nested sub query, in which case we apply the same idea one more time, leading to an evaluation strategy with several levels of nested loops.

(Q 13) Find the names of sailors who have reserved a red boat.

SQL> SELECT S.sname FROM Sailors S WHERE S.sid IN (SELECT R.sid FROM Reserves R WHERE R. bid IN (SELECT B.bid FROM Boats B WHERE B.color = 'red'));

To find the names of sailors who have not reserved a red boat, we replace the outermost occurrence of IN by NOT IN, as illustrated in the next query.

(Q14) Find the names of sailors who have not reserved a red boat.

SQL> SELECT S.sname FROM Sailors S WHERE S.sid NOT IN (SELECT R.sid FROM Reserves R WHERE R.bid IN (SELECT B.bid FROM Boats B WHERE B.color = 'red'));

4.2. Correlated Nested Queries:

In the nested queries seen thus far, the inner sub query has been completely independent of the outer query. In general, the inner sub query could depend on the row currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more.

(Q15) Find the names of sailors who have reserved boat number 103.

SQL> SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid = 103 AND R.sid = S.sid);

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each Sailor row S, we test whether the set of Reserves rows R such that R.bid = 103 AND S.sid = R.sid is nonempty. If so, sailor S has reserved boat 103, and we retrieve the name. The sub query clearly depends on the current row S and must be re-evaluated for each row in Sailors. The occurrence of S in the sub query (in the form of the literal S.sid) is called a correlation, and such queries are called correlated queries.

This query also illustrates the use of the special symbol * in situations where all we want to do is to check that a qualifying row exists, and do not really want to retrieve any columns from the row. This is one of the two uses of * in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation.

As a further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat.

4.3. Set Comparison operators:

We have already seen the set-comparison operators EXISTS, IN and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<=, =, <>, >=, >}. (SOME is also available, but it is just a synonym for ANY).

(Q16) Find sailors whose rating is better than some sailor called Horatio.

SQL> SELECT S.sid FROM Sailors S WHERE S.rating > ANY (SELECT S2.rating FROM Sailors S2 WHERE S2.sname = 'Horatio');

(Q17) Find sailors whose rating is better than every sailor called Horatio.

SQL> SELECT S.sid FROM Sailors S WHERE S.rating > ALL (SELECT S2.rating FROM Sailors S2 WHERE S2.sname = 'Horatio');

The another example of illustrating ALL operator to find out sailors who had highest rating with the following example.

(Q18) Find the Sailors with the highest rating.

SQL> SELECT S.sid FROM Sailors S WHERE S.rating >= ALL (SELECT S2.rating FROM Sailors S2);

5. AGGREGATE OPERATORS:

In addition to simply retrieving data, we often want to perform some computation or summarization. SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing aggregate values such as MIN and SUM.

These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.

2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.

3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.

4. MAX (A): The maximum value in the A column.

5. MIN (A): The minimum value in the A column.

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL does not preclude this).

(Q 19) Find the average age of all sailors.

SQL> SELECT AVG (S.age) FROM Sailors S;

On instance S3 of Sailors table, the average age of sailors is 37.4.

(Q 20) Find the average age of sailors with a rating of 10.

SQL> SELECT AVG (S.age) FROM Sailors S WHERE S.rating = 10;

(Q 21) Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

SQL> SELECT S.sname, MAX (S.age) FROM Sailors S;

(OR)

SQL> SELECT S.sname, S.age FROM Sailors S WHERE S.age = (SELECT MAX (S2.age) FROM Sailors S2);

We can count the number of sailors using COUNT. This exarnple illustrates the use of * as an argument to COUNT, which is useful when we want to count all rows.

(Q 22) Count the number of sailors.

SQL> SELECT COUNT (*) FROM Sailors S;

(Q 23) Count the number of different sailor names.

SQL> SELECT COUNT (DISTINCT S.sname) FROM Sailors S;

(Q 24) Find the names of sailors who are older than the oldest sailor with a rating of 10.

SQL> SELECT S.sname FROM Sailors S WHERE S.age > (SELECT MAX (S2.age) FROM Sailors S2 WHERE S2.rating = 10);

5.1. The GROUP BY and HAVING Clauses:

Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). For example, consider the following query.

(Q 25) Find the age of the youngest sailor for each rating level.

If we know that ratings are integers in the range 1 to l0, we could write 10 queries of the form:

SQL> SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i;

Where i = 1,2, ... ,10 writing 10 such queries is tedious. More important, we may not know what rating levels exist in advance.

To write such queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause. In fact, the extension also includes an optional HAVING clause that can be used to specify qualifications over groups.

The general form of an SQL query with these extensions is:

SELECT [DISTINCT] select-list

FROM from-list

WHERE qualification

GROUP BY grouping-list

HAVING group-qualification

Using the GROUP BY clause, we can write (Q 25) as follows:

SQL> SELECT S.rating, MIN (S.age) FROM Sailors S GROUP BY S.rating;

Let us consider some important points concerning the new clauses:

- The select-list in the SELECT clause consists of

  (1) a list of column names and

  (2) a list of terms having the form aggop (column-name) AS newname.

We already saw AS used to rename output columns. Columns that are the result of aggregate operators do not already have a column name, and therefore giving the column a name with AS is especially useful. Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one group, which is a collection of rows that agree on the values of columns in grouping list. In general, if a column appears in list (1), but not in grouping-list, there can be multiple rows within a group that have different values in this column, and it is not clear what value should be assigned to this column in an answer row. We can sometimes use primary key information to verify that a column has a unique value in all rows within each group. For example, if the grouping-list contains the primary key of a table in the from-list, every column of that table has a unique value within each group. In SQL:1999, such columns are also allowed to appear in part (1) of the select-list.

- The expressions appearing in the group-qualification in the HAVING clause must have a single value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. To satisfy this requirement in SQL-92, a column appearing in the group qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list. In SQL:1999, two new set functions have been introduced that allow us to check whether every or any row in a group satisfies a condition; this allows us to use conditions similar to those in a WHERE clause.
- If GROUP BY is omitted, the entire table is regarded as a single group.

Let us explain the semantics of such a query through an example.

(Q 26) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

SQL> SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18 GROUP BY S.rating HAVING COUNT (*) > 1;

SQL:1999 has introduced two new set functions, EVERY and ANY. To illustrate these functions, we can replace the HAVING clause in our example by HAVING COUNT (*) > 1 AND EVERY (S.age <= 60);

In other words, we can write the same query in the following format:

SQL> SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18 AND S.age <= 60 GROUP BY S.rating HAVING COUNT (*) > 1;

5.2. More Examples of Aggregate Queries:

(Q 27) For each red boat, find the number of reservations for this boat.

SQL> SELECT B.bid, COUNT (*) AS reservationcount FROM Boats B, Reserves R WHERE R.bid = B.bid AND B.color = 'red' GROUP BY B.bid;

The above query specification is illegal, due to that the same query can be define in alternative way as follows:

SQL> SELECT B.bid, COUNT (*) AS reservationcount FROM Boats B, Reserves R WHERE R.bid = B.bid GROUP BY B.bid HAVING B.color = 'red';

(Q 28) Find the average age of sailors for each rating level that has at least two sailors.

SQL> SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S GROUP BY S.rating HAVING COUNT (*) > 1;

(Q 29) Find the average age of sailors who are of voting age (i.e. at least 18 years old) for each rating level that has at least two sailors.

SQL> SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S WHERE S. age >= 18 GROUP BY S.rating HAVING 1 < (SELECT COUNT (*) FROM Sailors S2 WHERE S.rating = S2.rating);

(Q 30) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such sailors.

SQL> SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S WHERE S. age> 18 GROUP BY S.rating HAVING 1 < (SELECT COUNT (*) FROM Sailors S2 WHERE S.rating = S2.rating AND S2.age >= 18);

(Q 31) Find those ratings for which the average age of sailors is the minimum overall ratings.

We use this query to illustrate that aggregate operations cannot be nested. One might consider writing it as follows:

SQL> SELECT S.rating FROM Sailors S WHERE AVG(S.age) = (SELECT MIN (AVG (S2.age)) FROM Sailors S2 GROUP BY S2.rating);

6. NULL VALUES:

Thus far, we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a rating column, what row should we insert for Dan?

That is needed here is a special value that denotes unknown. Suppose the Sailor table definition was modified to include a maiden-name column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the maiden-name column is inapplicable. Again, what value do we include in this column for the row representing Dan? SQL provides a special column value called null to use in such situations.

We use null when the column value is either unknown or inapplicable. Using our Sailor table definition, we might enter the row (98. Dan, null, 39) to represent Dan. The presence of null values complicates many issues, and we consider the impact of null values on SQL in this section.

6.1. Comparisons Using Null Values:

Consider a comparison such as rating = 8. If this is applied to the row for Dan, is this condition true or false?

Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons rating> 8 and rating < 8 as well. Perhaps less obviously, if we compare two null values using <, >, =, and so on, the result is always unknown. For example, if we have null in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is null; for example, we can say rating IS NULL, which would evaluate to true on the row representing Dan. We can also say rating IS NOT NULL, which would evaluate to false on the row for Dan.

6.2 Logical Connectives AND, OR, and NOT:

Now, what about Boolean expressions such as rating = 8 OR age < 40 and raing = 8 AND age < 40?

Considering the row for Dan again, because age < 40, the first expression evaluates to true regardless of the value of rating, but what about the second? We can only say unknown. But this example raises an important point once we have null values, we must define the logical operators AND, OR, and NOT using a three-valued logic in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown &., follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, OR evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, AND evaluates to true.)

6.3 Impact on SQL Constructs:

Boolean expressions arise in many contexts in SQL, and the impact of null values must be recognized. For example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of null values, any row that evaluates to false or unknown is eliminated. Eliminating rows that evaluate to unknown has a subtle but significant impact on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of null values is the definition of when two rows in a relation instance are regarded as duplicates. The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain null. Contrast this definition with the fact that if we compare two null values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly. As expected, the arithmetic operations +, -, *, and / all return null if one of their arguments is null. However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles null values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard null values--thus SUM cannot be understood as just the addition of all values in the (multi)set of values that it is applied to; a preliminary step of discarding all null values must also be accounted for. As a special case, if one of these operators-other than COUNT is applied to only null values, the result is again null.

6.4 Outer Joins:

Some interesting variants of the join operation that rely on null values, called outer joins, are supported in SQL.

Consider the join of two tables, say Sailors $\bowtie_c$ Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition **c** do not appear in the result. In an outer join, on the other hand, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned null values. In fact, there are several variants of the outer join idea.

In a left outer join, Sailor rows without a matching Reserves row appear in the result, but not vice versa.

In a right outer join, Reserves rows without a matching Sailors row appear in the result, but not vice versa.

In a full outer join, both Sailors and Reserves rows without a match appear in the result. (Of course, rows with a match always appear in the result, for all these variants, just like the usual joins, sometimes called inner joins, presented in Chapter 4.)

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query lists (sid, bid) pairs corresponding to sailors and boats they have reserved:

SQL> SELECT S.sid, R.bid FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R;

The NATURAL keyword specifies that the join condition is equality on all common attributes (in this example, sid).

6.5 Disallowing Null Values:

We can disallow null values by specifying NOT NULL as part of the field definition; for example, sname CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on null values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

7. COMPLEX INTEGRITY CONSTRAINTS IN SQL:

In this section we discuss the specification of complex integrity constraints that utilize the full power of SQL queries. The features discussed in this section complement the integrity constraint features of SQL presented in Unit 2.

7.1 Constraints over a Single Table:

We can specify complex constraints over a single table using table constraints, which have the form CHECK conditional-expression. For example, to ensure that rating must be an integer in the range 1 to 10, we could use:

SQL> CREATE TABLE Sailors (sid INTEGER, sname CHAR(10), rating INTEGER, age REAL, PRIMARY KEY (sid), CHECK (rating >= 1 AND rating <= 10));

7.2 Domain Constraints and Distinct Types:

A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints.

SQL> CREATE DOMAIN ratingval INTEGER DEFAULT 1 CHECK (VALUE >= 1 AND VALUE <= 10);

INTEGER is the underlying, or source, type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema declaration, for example:

rating ratingval;

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation and no value is entered for this column in an inserted tuple, the default value 1 associated with ratingval is used. SQL support for the concept of a domain is limited in an important respect. For example, we can define two domains called SailorId and BoatId, each using INTEGER as the underlying type. The intent is to force a comparison of a SailorId value with a BoatId value to always fail (since they are drawn from different domains); however, since they both have the same base type, INTEGER, the comparison will succeed in SQL.

This problem is addressed through the introduction of distinct types in SQL:1999.

CREATE TYPE ratingtype AS INTEGER

This statement defines a new distinct type called ratingtype, with INTEGER as its source type. Values of type ratingtype can be compared with each other, but they cannot be compared with values of other types. In particular, ratingtype values are treated as being distinct from values of the source type, INTEGER. We cannot compare them to integers or combine them with integers (e.g., add an integer to a ratingtype value). If we want to define operations on the new type, for example, an average function, we must do so explicitly; none of the existing operations on the source type carryover.

7.3. Assertions: ICs over Several Tables:

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold only if the associated table is nonempty. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired. To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table. As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100. (This condition might be required, say, to qualify as a 'small' sailing club.) We could try the following table constraint:

```
SQL> CREATE TABLE Sailors ( sid INTEGER,
                            sname CHAR(10) ,
                            rating INTEGER,
                            age REAL, PRIMARY KEY (sid),
                           CHECK ( rating >= 1 AND rating <= 10)
CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S ) + ( SELECT COUNT (B. bid) FROM
Boats B ) < 100 ))
```

This solution suffers from two drawbacks. It is associated with Sailors, although it involves Boats in a completely symmetric way. More important, if the Sailors table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in Boats. We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes cumbersome.

The best solution is to create an assertion, as follows:

SQL> CREATE ASSERTION smallClub CHECK ((SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B. bid) FROM Boats B) < 100);

8. TRIGGERS AND ACTIVE DATABASES:

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database.

A trigger description contains three parts:

- Event: A change to the database that activates the trigger.
- Condition: A query or test that is run when the trigger is activated.
- Action: A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the event specification. An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.

A condition in a trigger can be a true/false or a query. A query is interpreted as true if the answer set is nonempty and false if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed. A trigger action can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute Hew queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit) or call host-language procedures.

An important issue is when the action part of a trigger executes in relation to the statement that activated the trigger.

For example, a statement that inserts records into the Students table may activate a trigger that is used to maintain statistics on how many students younger than 18 are inserted at a time by a typical insert statement. Depending on exactly what the trigger does, we may want its action to execute before changes are made to the Students table or afterwards: A trigger that initializes a variable used to count the number of qualifying insertions should be executed before, and a trigger that executes once per qualifying inserted record and increments the variable should be executed after each record is inserted (because we may want to examine the values in the new record to determine the action).

8.1 Examples of Triggers in SQL:

The examples, written using Oracle Server syntax for defining triggers, illustrate the basic concepts behind triggers. The trigger called **initcount** initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called **incr_count** increments the counter for each inserted tuple that satisfies the condition age < 18. One of the example triggers executes before the activating statement, and the other example executes after it. A trigger can also be scheduled to execute instead of the activating statement; or in deferred fashion, at the end of the transaction containing the activating statement; or in asynchronous fashion, as part of a separate transaction. The example in following illustrates another point about trigger execution: A user must be able to specify whether a trigger is to be executed once per modified record or once per activating statement. If the action depends on individual changed records, for example, we have to examine the age field of the inserted Students record to decide whether to increment the count, the triggering event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a row-level trigger. On the other hand, the **initcount** trigger is executed just once per INSERT statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a statement-level trigger.

```
CREATE TRIGGER initcount BEFORE INSERT ON Students /* Event */
DECLARE count INTEGER: BEGIN /* Action */
count := 0;
END
CREATE TRIGGER incr_count AFTER INSERT ON Students /* Event */
WHEN (new.age < 18) /* Condition; 'new' is just-inserted tuple */
FOR EACH ROW BEGIN /* Action; a procedure in Oracle's PL/SQL syntax */
count := count + 1;
END
```

UNIT-IV
SCHEMA REFINEMENT AND NORMALIZATION

1. Introduction:

Redundancy is at the root of several problems associated with relational schemas: redundant storage, insert/delete/update anomalies Integrity constraints, in particular functional dependencies, can be used to identify schemas with such problems and to suggest refinements.

Main refinement technique: decomposition (replacing ABCD with, say, AB and BCD, or ACD and ABD). Decomposition should be used judiciously:

Is there reason to decompose a relation?

What problems (if any) does the decomposition cause?

2. Problems Caused by Redundancy:

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is 10 updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set.

Ex: Hourly_Emps(ssn, name, lot, rating, hourly wages, hours worked)

The key for Hourly_Emps is ssn. In addition, suppose that the hourly wages attribute is determined by the rating attribute. That is, for a given rating value, there is only one permissible hourly wages value. This IC is an example of a functional dependency. It leads to possible redundancy in the relation Hourly_Emps.

3. Use of Decomposition:

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (ICs) can be used to identify such situations and to suggest revetments to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of smaller relations. Each of the smaller relations contains a subset of the attributes of the original relation. We refer to this process as decomposition of the larger relation into the smaller relations. We can deal with the redundancy in Hourly_Emps by decomposing it into two relations:

Hourly_Emps2(ssn, name, lot, rating, hours worked)

Wages(rating, hourly wages)

4. Problems related to Decomposition:

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?

2. What problems (if any) does a given decomposition cause?

To help with the first question, several normal forms have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

4.1. FUNCTIONAL DEPENDENCIES (FDs):

A Functional Dependency (FD) X-> Y (read as X determines Y) ($X \subseteq R$, $Y \subseteq R$) holds over relation R if, for every allowable instance r of R:

$$t1 \in r,\ t2 \in r,\ \pi_X(t1) = \pi_X(t2) \text{ implies } \pi_Y(t1) = \pi_Y(t2)$$

i.e., given two tuples in r, if the X values agree, then the Y values must also agree. (X and Y are sets of attributes).

An FD is a statement about all allowable relations. It must be identified based on semantics of application. Given some allowable instance r1 of R, we can check if it violates some FD f, but we cannot tell if f holds over R.

K is candidates key for R means that K->R. However, K-> R does not require K to be minimal.

Eg: Consider the schema:

Student (studName, rollNo, gender, dept, hostelName, roomNo)

Since rollNois a key**, rollNo → {studName, gender, dept, hostelName, roomNo}**

Suppose that each student is given a hostel room exclusively, then

hostelName, roomNo → rollNo

Suppose boys and girls are accommodated in separate hostels, then **hostelName → gender**

FDs are additional constraints that can be specified by designers.

An FD X →Y where Y ⊆ X - called a trivial FD, it always holds good.

An FD X →Y where Y ⊈ X -non -trivial FD

An FD X →Y where X ∩Y = Ø -completely non-trivial FD

Example: Constraints on Entity Set

Consider relation obtained from Hourly_Emps:

Hourly_Emps (ssn, name, lot, rating, hrly_wages, hrs_worked)

Notation: We will denote this relation schema by listing the attributes: SNLRWH

This is really the set of attributes {S, N, L, R, W, H}.

Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH).

Some FDs on Hourly_Emps:

- ssn is the key: S-> SNLRWH
- rating determines hrly_wages: R-> W

4.2. Reasoning about Functional Dependencies (FDs):

Given some FDs, we can usually infer additional FDs:

ssn → did, did → lot implies ssn → lot

An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.

F^+ = closure of F is the set of all FDs that are implied by F.

Armstrong's Axioms (X, Y, Z are sets of attributes):

1. **Reflexivity:** If X→Y, then Y→ X
2. **Augmentation:** If X→Y, then X→Z Y→Z for any Z
3. **Transitivity:** If X→ Y and Y→ Z, then X→Z

These are sound and complete inference rules for FDs.

Couple of additional rules (that follow from AA):

Union: If X→Y and X→ Z, then X→ YZ

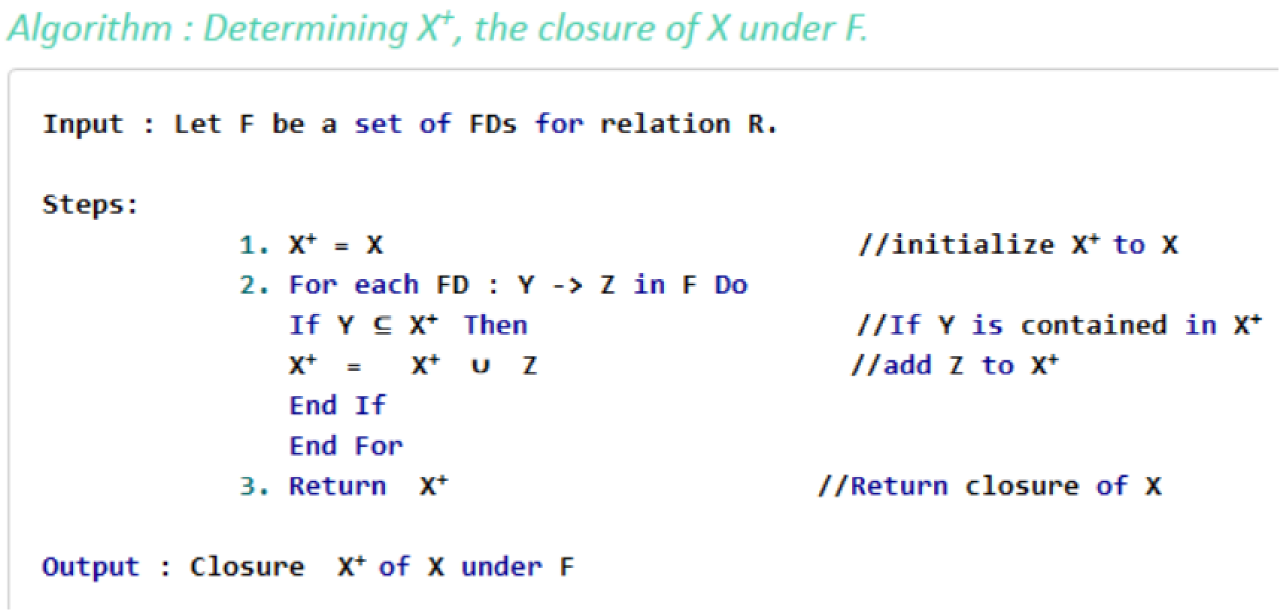
Decomposition: If X→ YZ, then X→ Y and X→ Z

Attribute closure: Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.

NOTE: To find attribute closure of an attribute set-

1)add elements of attribute set to the result set.

2) Recursively add elements to the result set which can be functionally determined from the elements of result set.



Algorithm : Determining X^+, the closure of X under F.

```
Input : Let F be a set of FDs for relation R.

Steps:
            1. X⁺ = X                               //initialize X⁺ to X
            2. For each FD : Y -> Z in F Do
               If Y ⊆ X⁺  Then                      //If Y is contained in X⁺
               X⁺  =   X⁺  ∪  Z                     //add Z to X⁺
               End If
               End For
            3. Return  X⁺                        //Return closure of X

Output : Closure  X⁺ of X under F
```

Prime and non-prime attributes

Attributes which are parts of any candidate key of relation are called as prime attribute, others are non-prime attributes.

Candidate Key: Candidate Key is minimal set of attributes of a relation which can be used to identify a tuple uniquely.

Consider student table: student(sno, sname,sphone,age)

we can take sno as candidate key.

we can have more than 1 candidate key in a table.

types of candidate keys:

1. simple(having only one attribute)

2. composite(having multiple attributes as candidate key)

Super Key: Super Key is set of attributes of a relation which can be used to identify a tuple uniquely.

Adding zero or more attributes to candidate key generates super key.

- A candidate key is a super key but vice versa is not true.

• Consider student table: student(sno, sname,sphone,age) we can take sno, (sno, sname) as super key.

Finding candidate keys problems:

Example 1: Find candidate keys for the relation R(ABCD) having following FD's

$AB \rightarrow CD$, $C \rightarrow A$, $D \rightarrow A$

Solution:

$AB^+ = \{ABCD\}$ A and B are prime attributes

$C \rightarrow A$ replace A by c

$BC^+ = \{ABCD\}$ A and C are prime attributes ($A^+ = A^+ = \{AC\}$)

$D \rightarrow B$ replace B by D

$AD^+ = \{ABCD\}$ A and D are prime attributes ($D^+ = \{BD\}$)

$CD^+ = \{ABCD\}$ (replacing A by C in AD)

AB, BC, CD, AD are candidate keys.

Example 2: Find candidate keys for R(ABCDE) having following FD's

$A \rightarrow BC, CD \rightarrow E, B \rightarrow D, E \rightarrow A$

Solution:

$A^+ = \{ABCDE\}$ A is candidate key and prime attribute

$E \rightarrow A$ so replace A by E

$E^+ = \{ABCDE\}$ E is candidate key and prime attribute

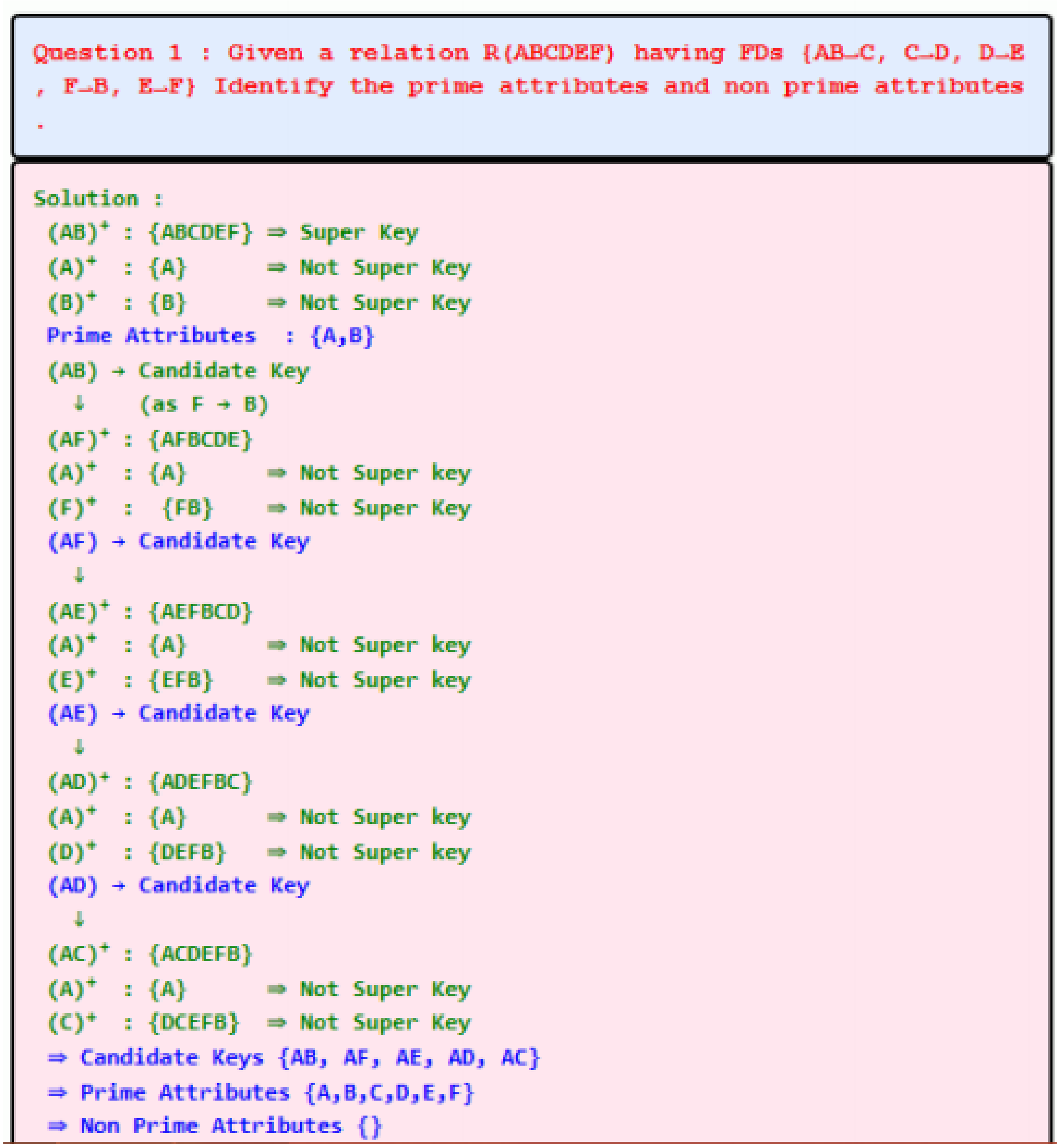
$CD \rightarrow E$ replace E by CD

$CD^+ = \{ABCDE\}$ ($C^+ = C$ and $D^+ = D$) no proper subset of CD is superkey. so CD is candidate key

$B \rightarrow D$

$BC^+ = \{ABCDE\}$ ($B^+ = BD$) BC is candidate key

A, E, CD, BC are candidate keys



Question 1 : Given a relation R(ABCDEF) having FDs {AB→C, C→D, D→E , F→B, E→F} Identify the prime attributes and non prime attributes .

Solution :

$(AB)^+$: {ABCDEF} ⇒ Super Key
$(A)^+$: {A} ⇒ Not Super Key
$(B)^+$: {B} ⇒ Not Super Key
Prime Attributes : {A,B}
(AB) → Candidate Key
↓ (as F → B)
$(AF)^+$: {AFBCDE}
$(A)^+$: {A} ⇒ Not Super key
$(F)^+$: {FB} ⇒ Not Super Key
(AF) → Candidate Key
↓
$(AE)^+$: {AEFBCD}
$(A)^+$: {A} ⇒ Not Super key
$(E)^+$: {EFB} ⇒ Not Super key
(AE) → Candidate Key
↓
$(AD)^+$: {ADEFBC}
$(A)^+$: {A} ⇒ Not Super key
$(D)^+$: {DEFB} ⇒ Not Super key
(AD) → Candidate Key
↓
$(AC)^+$: {ACDEFB}
$(A)^+$: {A} ⇒ Not Super Key
$(C)^+$: {DCEFB} ⇒ Not Super Key
⇒ Candidate Keys {AB, AF, AE, AD, AC}
⇒ Prime Attributes {A,B,C,D,E,F}
⇒ Non Prime Attributes {}

Normalization: Normalization is a process of designing a consistent database with minimum redundancy which support data integrity by grating or decomposing given relation into smaller relations preserving constraints on the relation. Normalization removes data redundancy and it will help in designing a good data base which involves a set of normal forms as follows –

1) First normal form (1NF)

2) Second normal form (2NF)

3) Third normal form (3NF)

4) Boyce coded normal form (BCNF)

5) Fourth Normal Form (4NF)

6) Fifth Normal Form (5NF)

The main goal of Database Normalization is to restructure the logical data model of a database to eliminate redundancy, organize data efficiently and reduce the potential for data anomalies. How to take a raw collection of data and break it up into more logical units or tables, in order to reduce the occurrence of redundant data in the database?. This process of reducing data redundancy is referred to as Normalization. Normalization is a body of rules addressing analysis and conversion of data structures into relations that exhibit more desirable properties of internal consistency, minimal redundancy and maximum stability.

Normal Form is a state of a relation that result by decomposing that relation for a good design to avoid redundancy. The Normal Forms defined in Relational database theory represent guidelines for record design.

1. First Normal Form:

Definition:

1. A relation is in the Fisrt Normal Form, if it does not contain any repeating elements or groups.
2. A relation is in the First Normal Form only if all underlying domains contain only atomic values.

The requirements to satisfy the 1st NF:

- Each table has a primary key: minimal set of attributes which can uniquely identify a record.
- The values in each column of a table are atomic (No multi valued attributes allowed).
- There are no repeating groups: two columns do not store similar information in the same table.

First Normal Form excludes variable repeating fields and groups. This is not so much a design guideline as a matter of definition. Relational database theory doesn’t deal with records having a variable no. of fields.

Drawback of First Normal Form: The main drawback of First Normal Form is redundancy of data.

2. Second Normal Form:

Definition: A relation is in the Second Normal Form if it is in the First Normal Form and all non-key attributes are fully functionally dependent on the primary key. A relation is in 2NF if all the non-key attributes are dependent on all of the Key attributes.

A table is in 2nd Normal Form if:

It is in 1st Normal Form

It includes no partial dependencies.

The steps to convert a table to its 2nd Normal Form:

- Find and remove fields that are related to the only part of the key.
- Group the removed items in another table.
- Assign the new table with the key i.e. part of a whole composite key.

3. Third Normal Form:

Definition:

1. A relation is in the 3NF if it is in the 2NF and every non-key attribute is non-transitively dependent on the primary key.
2. A relation is in the 3NF if it is in the 2NF and every attribute is independent of all other non-key attributes.

SCHEMA REFINEMENT IN DATABASE DESIGN

We have seen how normalization can eliminate redundancy and discussed several approaches to normalizing a relation. We now consider how these ideas are applied in practice. Database designers typically use a conceptual design methodology, such as ER design, to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of PI)s and normalization techniques. In this section, we intimate the need for a schema refinement step following ER design. It is natural to ask whether we even need to decompose relations produced by translating an ER diagram. Should a good ER design not lead to a collection of relations free of redundancy problems? Unfortunately, ER design is a complex, subjective process, and certain constraints are not expressible in terms of ER diagrams. The examples in this section are intended to illustrate why decomposition of relations produced through ER design might be necessary

UNIT-V
OVERVIEW OF TRANSACTION MANAGEMENT

UNIT III

Overview Of Transaction Management : The ACID Properties, Transactions and Schedules, Concurrent Execution of transactions-Lock Based Concurrency Control, Performance of Locking, Transaction Support in SQL, Introduction to crash recovery, Concurrency Control: 2PL,serializabilityand recoverability, Introduction Lock Management, Lock Conversions, Dealing with Deadlocks, Concurrency control without locking.

Introduction to Transaction Management:

A transaction is defined as any one execution of a user program in a DBMS and differs from an execution of a program outside the DBMS. For performance reasons, a DBMS has to interleave the actions of several transactions. However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect upon the database) to some serial, or one-at-a-time, execution of the same set of transactions. How the DBMS handles concurrent executions is an important aspect of transaction management and is the subject of concurrency control. A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion. The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of crash recovery.

ACID PROPERTIES:

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard an execution of a user program, or transaction, as a series of reads and writes of database objects:

1. To read a database object, it is first brought into main memory from disk, and then its value is copied into a program variable.
2. To write a database object, an in-memory copy of the object is first modified and then written to disk.

Database `objects' are the units in which programs read or write information. The units could be pages, records, and so on, but this is dependent on the DBMS and is not central to the principles underlying concurrency control or recovery. We will consider a database to be a fixed collection of independent objects.

There are four important properties of transactions that a DBMS must ensure to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as atomic: either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions.
2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. This property is called consistency, and the DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

Consistency and Isolation:

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that when run to completion by itself against a `consistent' database instance, the transaction will leave the database in a 'consistent' state.

The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order.

Database consistency is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation, and transaction consistency.

Atomicity and Durability:

Transactions can be incomplete for three kinds of reasons. First, a transaction can be aborted, or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew. Second, the system may crash while one or more transactions are in progress. Third, a transaction may encounter an unexpected situation and decide to abort.

since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Thus a DBMS must find a way to remove the effects of partial transactions from the database, that is, it must ensure transaction atomicity: either all of a transaction's actions are carried out, or none are. A DBMS ensures transaction atomicity by undoing the actions of incomplete transactions. This means that users can ignore incomplete transactions
in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the log, of all writes to the database.

The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts. The DBMS component that ensures atomicity and durability is called the recovery Manager.

TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects. A transaction can also be defined as a set of actions that are partially ordered. That is, the relative order of some of the actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a list of actions. Further, to keep our notation simple, we'll assume that an object O is always read into a program variable that is also named O. We can therefore denote the action of a transaction T reading an object O as R_T (O); similarly, we can denote writing as W_T (O). When the transaction T is clear from the context, we will omit the subscript. In addition to reading and writing, each transaction must specify as its final action either commit (i.e., complete successfully) or abort (i.e., terminate and undo all the actions carried out thus far). Abort T denotes the action of T aborting, and Commit T denotes T committing.

A schedule is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. A schedule represents an actual or potential execution sequence. For example, the schedule in the following Figure shows an execution order for actions of two transactions T1 and T2. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions as seen by the DBMS. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on.

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

A Schedule involving two transactions

The schedule in figure does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a complete schedule. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved, that is, transactions are executed from start to finish one by one then it is called a serial schedule.

CONCURRENT EXECUTION OF TRANSACTIONS

Now that we've introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleaving should be allowed. In this section we consider what interleaving, or schedules, a DBMS should allow.

Motivation for Concurrent Execution

The schedule shown in above figure represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult, but it is necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases system throughput (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction leading to unpredictable delays in response time, or average time taken to complete a transaction.

Serializability

To begin with, we assume that the database designer has defined some notion of a consistent database state. For example, we can define a consistency criterion for a university database to be that the sum of employee salaries in each department should be less than 80 percent of the budget for that department. We require that each transaction must preserve database consistency; it follows that any serial schedule that is complete will also preserve database consistency. That is, when a complete serial schedule is executed against a consistent database, the result is also a consistent database.

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order.

Anomalies Associated with Interleaved Execution

We now illustrate three main ways in which a schedule involving two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state.

Two actions on the same data object conflict if at least one of them is write. The three anomalous situations can be described in terms of when the actions of two transactions T1 and T2 conflict with each other: in a write read (WR) conflict T2 reads a data object previously written by T1; we define read-write (RW) and write-write (WW) conflicts similarly.

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction T2 could read a database object A that has been modified by another transaction T1, which has not yet committed. Such a read is called a dirty read. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions T1 and T2, each of which, run alone, preserves database consistency:
The general problem illustrated here is that T1 may write some value into A that makes the database inconsistent. As long as T1 overwrites this value with a `correct' value of A before committing, no harm is done if T1 and T2 run in some serial order, because T2 would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Unrepeatable Reads (RW Conflicts)

The second way in which anomalous behavior could result is that a transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress. This situation causes two problems.
First, if T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. This situation could not arise in a serial execution of two transactions; it is called an unrepeatable read.

Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress. Even if T2 does not read the value of A written by T1, a potential problem exists as the following example illustrates.
Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction T1 sets their salaries to $1,000 and transaction T2 sets their salaries to $2,000. If we execute these in the serial order T1 followed by T 2, both receive the salary $2,000; the serial order T2 followed by T 1 gives each the salary $1,000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Notice that neither transaction reads a salary value before writing it, such a write is called a blind write.

Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions. Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A serializable schedule over a set S of transactions is a schedule whose e_ect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in S.
This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program T1 deducts $100 from account A, then (2) an interest deposit program T2 reads the current values of accounts A and B and adds 6 percent interest to each, then commits, and then (3) T1 is aborted.
If T2 had not yet committed, we could deal with the situation by cascading the abort of T1 and also aborting T2; this process would recursively abort any transaction that read data written by T2, and so on. But T2 has already committed, and so we cannot undo its actions! We say that such a schedule is unrecoverable. A recoverable schedule is one in which transactions commit only after (and if!) all

transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to avoid cascading aborts.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction T2 overwrites the value of an object A that has been modified by a transaction T1, while T1 is still in progress, and T1 subsequently aborts. All of T1's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before T1's changes. (We will look at the details of how a transaction abort is handled in Chapter 20.) When T1 is aborted, and its changes are undone in this manner, T2's changes are lost as well, even if T2 decides to commit. So, for example, if A originally had the value 5, then was changed by T1 to 6, and by T2 to T1, if T1 now aborts, the value of A becomes 5 again. Even if T2 commits, its change to A is inadvertently lost. A concurrency control technique called Strict 2PL, introduced.

LOCK-BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a locking protocol to achieve this. A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

18.4.1 Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules. The first rule is

(1) If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is:

(2) All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry, in effect the locking protocol allows only `safe' inter leavings of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released) and the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as ST (O) (respectively, XT (O)), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 18.2. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, T1 could change A from 10 to 20, then T2 (which reads the value 20 for A) could change B from 100 to 200, and then T1 would read the value 200 for B. If run serially, either T1 or T2 would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, T1 would obtain an exclusive lock on A _rst and then read and write A (Figure 18.4).
Then, T2 would request a lock on A. However, this request cannot be granted until T1 releases its exclusive lock on A, and the DBMS therefore suspends T2. T1 now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. T2's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure 18.5. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 18.6, which is permitted by the Strict 2PL protocol.

Deadlocks

Consider the following example: transaction T1 gets an exclusive lock on object A, T2 gets an exclusive lock on B, T1 requests an exclusive lock on B and is queued, and T2 requests an exclusive lock on A and is queued. Now, T1 is waiting for T2 to release its lock and T2 is waiting for T1 to release its lock! Such a cycle of transactions waiting for locks to be released is called a deadlock. Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations.

PERFORMANCE OF LOCKING

Lock-based schemes are designed to resolve conflicts between transactions and use two ba'3ic mechanisms: blocking and aborting. Both mechanisms involve a performance penalty: Blocked transactions may hold locks that force other transactions to wait, and aborting and restarting a transaction obviously wastes the work done thus far by that transaction. A deadlock represents an extreme instance of blocking in which a set of transactions is forever blocked unless one of the deadlocked transactions is aborted by the DBMS.
In practice, fewer than 1%of transactions are involved in a deadlock, (there are relatively few aborts. Therefore, the overhead of locking comes primarily from delays due to blocking. Consider how blocking delays affect throughput. Consider how blocking delays affect throughput.
The first few transactions are unlikely to conflict, and throughput rises in proportion to the number of active transactions. As more and more transactions execute concurrently on the same number of database objects, the likelihood of their blocking each other goes up. Thus, delays due to blocking increase with the number of active transactions, and throughput increases more slowly than the number of active transactions. In fact, there comes a point when adding another active transaction actually reduces throughput; the new transaction is blocked and effectively competes with (and blocks) existing transactions. DBA should monitor the fraction of blocked transactions to see if the system is at risk of thrashing.
Throughput can be increased in three ways:

- By locking the smallest sized objects possible (reducing the likelihood that two transactions need the same lock).
- By reducing the time that transaction hold locks (so that other transactions are blocked for a shorter time).
- By reducing hot spots. A hot spot is a database object that is frequently accessed and modified, and causes a lot of blocking delays. Hot spots can significantly affect performance.

The granularity of locking is largely determined by the database system's implementation of locking, and application programmers and the DBA have little control over it.

TRANSACTION SUPPORT IN SQL

We have thus far studied transactions and transaction management using an abstract model of a transaction as a sequence of read, write, and abort/commit actions. We now consider what support SQL provides for users to specify transaction-level behavior.

Creating and Terminating Transactions

A transaction is automatically started when a user executes a statement that accesses either the database or the catalogs, such as a SELECT query, an UPDATE command, or a CREATE TABLE statement. Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a COMMIT command or a ROLLBACK (the SQL keyword for abort) command.

In SQL:1999, two new features are provided to support applications that involve long-running transactions, or that must run several transactions one after the other. To understand these extensions, recall that all the actions of a given transaction are executed in order, regardless of how the actions of different transactions are interleaved. We can think of each transaction as a sequence of steps.

The first feature, called a save point, allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results. This can be accomplished by defining save points.

In a long-running transaction, we may want to define a series of save points. The save point command allows us to give each save point a name:

SAVEPOINT (save point name)

A subsequent rollback command can specify the save point to roll back to

ROLLBACK TO SAVEPOINT (save point name)

What Should We Lock?

Until now, we have discussed transactions and concurrency control in tenus of an abstract model in which a database contains a fixed collection of objects, and each transaction is a series of read and write operations on individual objects.

An important question to consider in the context of SQL is what the DBMS should treat as an object when setting locks for a given SQL statement (that is part of a transaction).

Consider the following query:

SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.rating = 8

Suppose that this query runs as part of transaction T1 and an SQL statement that modifies the age of a given sailor, say Joe, with rating=8 runs a-s part of transaction T2. What 'objects' should the DBMS lock when executing these transactions? Intuitively, we must detect a conflict between these transactions.

The DBMS could set a shared lock on the entire Sailors table for T1 and setan exclusive lock on Sailors for T2, which would ensure that the two transactions are executed in a serializable manner. However, this approach yields low concurrency, and we can do better by locking smaller objects, reflecting what each transaction actually accesses. Thus, the DBMS could set a shared lock on every row with mting=8 for transaction T1 and set an exclusive lock on just the row for the modified tuple for transaction T2. Now, other read-only transactions that do not involve nding=8 rows can proceed without waiting for T1 or T2.

As this example illustrates, the DBMS can lock objects at different granularities: we can lock entire tables or set row-level locks. The latter approach is taken in current systems because it offers much better performance.

Transaction Characteristics in SQL

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The diagnostics size determines the number of error conditions that can be recorded;
If the access mode is READ ONLY, the transaction is not allowed to modify the databclse. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE or transactions with READ ONLY access mode. only shared locks need to be obtained, thereby increasing concurrency.

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

The isolation level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transaction's uncommitted changes.
Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE.
The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
When a transaction is started, the default is SERIALIZABLE and READ WRITE.

INTRODUCTION TO CRASH RECOVERY

The recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of committed transactions survive system crashes, (e.g., a core dump caused by a bus error) and media failures (e.g., a disk is corrupted).Then a DBMS is restarted after crashes. the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what
it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.
The transaction manager of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol. For simplicity of exposition, we make the following assumption:
Atomic Writes: Writing a page to disk is an atomic action.

Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:
1. Can the changes made to an object 0 in the buffer pool by a transaction T be written to disk before T commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the frame containing 0; of course, this page must have been unpinned by T. If such writes are allowed, we say that a steal approach is used. (Informally, the second transaction 'steals' a frame from T.)

2. When a transaction cOl1units, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so. we say that a force approach is used.
From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steaL force approach. If a no-steal approach is used, we do not have to undo the changes of an aborted transaction (because these dumges have not been written to disk) l and if a force approach is used, we do not have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).
However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs.

Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions to enable it to perform its task in the event of a failure. In particular, a log of all modifications to the database is saved on stable storage, which is guaranteed6 to survive crashes and media failures.
Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.
The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log and written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.
The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process (while making sure that any log entries that describe changes these pages are written to disk first, i.e., following the WAL protocol). A process called check pointing, which saves information about active transactions and dirty buffer pool pages, also helps reduce the time taken to recover from a crash.

Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases. In the Analysis phase, it identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash. In the Redo phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash. Finally, in the Undo phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.
When the recovery manager is invoked after a crash, restart proceeds in three phases:
1. Analysis: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.
2. Redo: Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
3. Undo: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

2PL, SERIALIZABILITY, AND RECOVERABILITY

In this section, we consider how locking protocols guarantee some important properties of schedules; namely, serializability and recoverability. Two schedules are said to be conflict equivalent if they involve the (same set of) actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way. Two actions conflict if they operate on the same data object and at least one of them is a write. The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of non-conflicting operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database.
Indeed, because they order all pairs of conflicting operations in the same way, we can obtain one of them from the other by repeatedly swapping pairs of non-conflicting actions, that is, by swapping pairs of actions whose relative order does not alter the outcome.
A schedule is conflict serializable if it is conflict equivalent to some serial schedule. Every conflict serializable schedule is serializable, if we assume that the set of items in the database does not grow or shrink; that is, values can be modified but items are not added or deleted. We will make this assumption for now and consider its consequences This schedule is equivalent to executing the transactions This schedule is equivalent to executing the transactions. It is useful to capture all potential conflicts between the transactions in a schedule in a precedence graph, also called a serializability graph. The precedence graph for a schedule S contains:
A node for each committed transaction in S.
An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.
1. A schedule S is conflict serializable if and only if its precedence graph is acyclic.
2. Strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic.
A widely studied variant of Strict 2PL, called Two-Phase Locking (2PL), relaxes the second rule of Strict 2PL to allow transactions to release locks before the end, that is, before the commit or abort action. For 2PL, the second rule is replaced by the following rule:
(2PL) (2) A transaction cannot request additional locks once it releases any lock.
Thus, every transaction has a `growing' phase in which it acquires locks, followed by a `shrinking' phase in which it releases locks. It can be shown that even (non-strict) 2PL ensures a cyclicity of the precedence graph and therefore allows only serializable schedules. Intuitively, an equivalent serial order of transactions is given by the order in which transactions enter their shrinking phase:
If T2 reads or writes an object written by T1, T1 must have released its lock on the object before T2 requested a lock on this object. Thus, T1 will precede T2. (A similar argument shows that T1 precedes T2 if T2 writes an object previously read by T1. A formal proof of the claim would have to show that there is no cycle of transactions that `precede' each other by this argument.)
A schedule is said to be strict if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits. Strict schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified objects. (See the last example in Section 18.3.4.) Strict 2PL improves upon 2PL by guaranteeing that every allowed schedule is strict, in addition to being conflict serializable. The reason is that when a transaction T writes an object under Strict 2PL, it holds the (exclusive) lock until it commits or aborts. Thus, no other transaction can see or modify this object until T is complete.

View Serializability

Conflict serializability is su_cient but not necessary for serializability. A more general su_cient condition is view serializability. Two schedules S1 and S2 over the same set of transactions|any transaction that appears in either S1 or S2 must also appear in the other|are view equivalent under these conditions:

1. If Ti reads the initial value of object A in S1, it must also read the initial value of A in S2.
2. If Ti reads a value of A written by Tj in S1, it must also read the value of A written by Tj in S2.
3. For each data object A, the transaction (if any) that performs the final write on A in S1 must also perform the final write on A in S2.

A schedule is view serializable if it is view equivalent to some serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true.

LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the lock manager. the data object identifier as the key. The DBMS also maintains a descriptive entry for each transaction in a transaction table, and among other things, the entry contains a pointer to a list of locks held by the transaction.

A lock table entry for an object which can be a page, a record, and so on, depending on the DBMS contains the following information: the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

Implementing Lock and Unlock Requests

According to the Strict 2PL protocol, before a transaction T reads or writes a database object O, it must obtain a shared or exclusive lock on O and must hold on to the lock until it commits or aborts. When a transaction needs a lock on an object, it issues a lock request to the lock manager:

1. If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one).
2. If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.
3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

When a transaction aborts or commits, it releases all its locks. When a lock on an object is released, the lock manager updates the lock table entry for the object and examines the lock request at the head of the queue for this object.

Atomicity of Locking and Unlocking

The implementation of lock and unlock commands must ensure that these are atomic operations. To ensure atomicity of these operations when several instances of the lock manager code can execute concurrently, access to the lock table has to be guarded by an operating system synchronization mechanism such as a semaphore. To understand why, suppose that a transaction requests an exclusive lock. The lock manager checks and finds that no other transaction holds a lock on the object and therefore decides to grant the request.

Other Issues: Latches, Convoys

In addition to locks, which are held over a long duration, a DBMS also supports short-duration latches. Setting a latch before reading or writing a page ensures that the physical read or write operation is atomic; otherwise, two read/write operations right conflict if the objects being locked do not correspond to disk pages (the units of I/O). Latches are unset immediately after the physical read or write operation is completed.

We concentrated thus far on how the DBMS schedules transactions based on their requests for locks. This interleaving interacts with the operating system's scheduling of processes' access to the CPU and can lead to a situation called a convoy, where most of the CPU cycles are spent on process switching. The problem is that a transaction T holding a heavily used lock may be suspended by the operating system. Until T is returned, every other transaction that needs this lock is queued. Such queues, called convoys, can quickly become very long; a convoy, once formed, tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preemptive scheduling.

LOCK CONVERSIONS

A transaction may need to acquire an exclusive lock on an object for which it already holds a shared lock. For example, a SQL update statement could result in shared locks being set on each row in a table. If a row satisfies the condition (in the WHERE clause) for being updated, an exclusive lock must be obtained for that row.

Such a lock upgrade request must be handled specially by granting the exclusive lock immediately if no other transaction holds a shared lock on the object and inserting the request at the front of the queue other\vise. The rationale for favoring the transaction thus is that it already 1101ds a shared lock on the object and queuing it behind. Another transaction that wants an exclusive lock on the Same object causes both a deadlock. Unfortunately, while favoring lock upgrades helps, it does not prevent deadlocks caused by two conflicting upgrade requests. For example, if two transactions that hold a shared lock on an object both request an upgrade to an exclusive lock, this leads to a deadlock.

A better approach is to avoid the need for lock upgrades altogether by obtaining exclusive locks initially and downgrading to a shared lock once it is clear that this is sufficient.

The downgrade approach reduces concurrency by obtaining write locks in some cases where they are not required. On the whole, however, it improves throughput by reducing deadlocks. Concurrency can be increased by introducing a new kind of lock, called an update lock, that is compatible with shared locks but not other update and exclusive locks. By setting an update lock initially, rather than exclusive locks, we prevent conflicts with other read operations. Once we are sure we need not update the object, we can downgrade to a shared lock. If we need to update the object, we must first upgrade to an exclusive lock. This upgrade does not lead to a deadlock because no other transaction can have an upgrade or exclusive lock on the object.

DEALING WITH DEADLOCKS

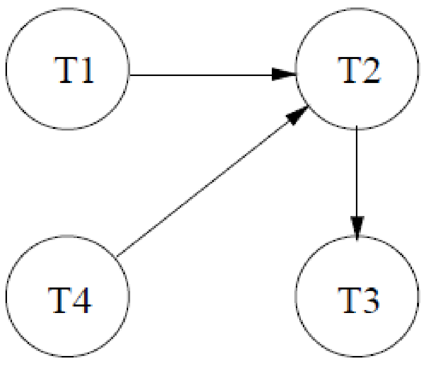
Deadlocks tend to be rare and typically involve very few transactions. In practice, therefore, database systems periodically check for deadlocks. When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them. The lock manager maintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from Ti to 'Tj if (and only if)Ti is waiting for 1) to release a lode The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests. A simple alternative to maintaining a waits-for graph is to identify deadlocks through a timeout mechanism: If a transaction has been waiting too long for a lock, we assume that it is in a deadlock cycle and abort it.

Deadlock Detection

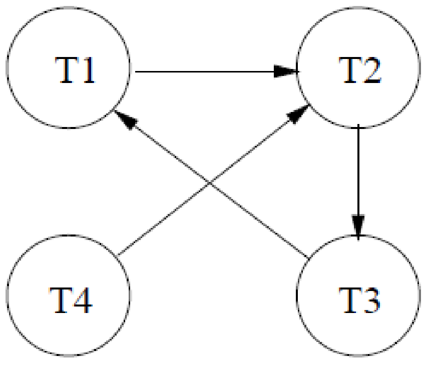
Deadlocks tend to be rare and typically involve very few transactions. This observation suggests that rather than taking measures to prevent deadlocks, it may be better to detect and resolve deadlocks as they arise. In the detection approach, the DBMS must periodically check for deadlocks.

$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	

When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them. The lock manager maintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from Ti to Tj if (and only if) Ti is waiting for Tj to release a lock. The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.



(a)



(b)

Deadlock Prevention

Empirical results indicate that deadlocks are relatively infrequent, and detection based schemes work well in practice. However, if there is a high level of contention for locks and therefore an increased likelihood of deadlocks, prevention based schemes could perform better. We can prevent deadlocks by giving each transaction a priority and ensuring that lower-priority transactions are not allowed to wait for higher-priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher is the transaction's priority; that is, the oldest transaction has the highest priority.
If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:

1. Wait-die: If Ti has higher priority, it is allowed to wait; otherwise, it is aborted.
2. Wound-wait: If Ti has higher priority, abort; otherwise, Ti waits.

In the wait-die scheme, lower-priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher-priority transactions never wait for lower-priority transactions. In either ease, no deadlock cycle develops.
The wait-die scheme is non-preemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs is never aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).
A variant of 2PL, called Conservative 2PL, can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will be no deadlocks, and, perhaps more important, that a transaction that already holds some locks will not block waiting for other locks.

CONCURRENCY CONTROLWITHOUT LOCKING

Locking is the most widely used approach to concurrency control in a DBMS, but it is not the only one. We now consider some alternative approaches.

Optimistic Concurrency Control

Locking protocols take a pessimistic approach to conflicts between transactions and use either transaction abort or blocking to resolve conflicts. In a system with relatively light contention for data objects, the overhead of obtaining locks and following a locking protocol must nonetheless be paid.
In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute. Transactions proceed in three phases:
1. Read: The transaction executes, reading values from the database and writing to a private workspace.
2. Validation: If the transaction decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, the transaction is aborted; its private workspace is cleared and it is restarted.
3. Write: If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.
Each transaction Ti is assigned a timestamp TS(Ti) at the beginning of its validation phase, and the validation criterion checks whether the timestamp-ordering of transactions is an equivalent serial order. For every pair of transactions Ti and Tj such that TS(Ti) < TS(Tj), one of the following conditions must hold:
1. Ti completes (all three phases) before Tj begins; or
2. Ti completes before Tj starts its Write phase, and Ti does not write any database object that is read by Tj; or
3. Ti completes its Read phase before Tj completes its Read phase, and Ti does not write any database object that is either read or written by Tj.
To validate Tj, we must check to see that one of these conditions holds with respect to each committed transaction Ti such that TS(Ti) < TS(Tj). Each of these conditions ensures that Tj's modifications are not visible to Ti.
Further, the first condition allows Tj to see some of Ti's changes, but clearly, they execute completely in serial order with respect to each other. The second condition allows Tj to read objects while Ti is still modifying objects, but there is no conflict because Tj does not read any object modified by Ti. Although Tj might overwrite some objects written by Ti, all of Ti's writes precede all of Tj's writes. The third condition allows Ti and Tj to write objects at the same time, and thus have even more overlap in time than the second condition, but the sets of objects written by the two transactions

cannot overlap. Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met.

Timestamp-Based Concurrency Control

In lock-based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained, and the lock protocol extends this ordering on actions to transactions, thereby ensuring serializability. In optimistic concurrency control, a timestamp ordering is imposed on transactions, and validation checks that all conflicting actions occurred in the same order.

Timestamps can also be used in another way: each transaction can be assigned a timestamp at startup, and we can ensure, at execution time, that if action ai of transaction Ti conflicts with action aj of transaction Tj, ai occurs before aj if TS(Ti) < TS(Tj).

If an action violates this ordering, the transaction is aborted and restarted. To implement this concurrency control scheme, every database object O is given a read timestamp RTS(O) and a write timestamp WTS(O). If transaction T wants to read object O, and TS(T) < WTS(O), the order of this read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted with a new, larger timestamp. If TS(T) > WTS(O), T reads O, and RTS(O) is set to the larger of RTS(O) and TS(T). (Note that there is a physical change, the change to RTS(O)|to be written to disk and to be recorded in the log for recovery purposes, even on reads. This write operation is a significant overhead.)

Observe that if T is restarted with the same timestamp, it is guaranteed to be aborted again, due to the same conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention: there, transactions were restarted with the same timestamp as before in order to avoid repeated restarts.

when transaction T wants to write object O:

1. If TS(T) < RTS(O), the write action conflicts with the most recent read action of O, and T is therefore aborted and restarted.
2. If TS(T) < WTS(O), a naive approach would be to abort T because its write action conflicts with the most recent write of O and is out of timestamp order. It turns out that we can safely ignore such writes and continue. Ignoring outdated writes is called the Thomas Write Rule.
3. Otherwise, T writes O and WTS(O) is set to TS(T).

Multi-version Concurrency Control

This protocol represents yet another way of using timestamps, assigned at startup time, to achieve serializability. The goal is to ensure that a transaction never has to wait to read a database object, and the idea is to maintain several versions of each database object, each with a write timestamp, and to let transaction Ti read the most recent version whose timestamp precedes TS(Ti). For an object, we must ensure that the object has not already been read by some other transaction Tj such that TS(Ti) < TS(Tj). If we allow Ti to write such an object, its change should be seen by Tj for serializability,
but obviously Tj, which read the object at some time in the past, will not see Ti's change.

To check this condition, every object also has an associated read timestamp, and whenever a transaction reads the object, the read timestamp is set to the maximum of the current read timestamp and the reader's timestamp. If Ti wants to write an object O and TS(Ti) < RTS(O), Ti is aborted and restarted with a new, larger timestamp.

Otherwise, Ti creates a new version of O, and sets the read and write timestamps of the new version to TS(Ti).

The drawbacks of this scheme are similar to those of timestamp concurrency control, and in addition there is the cost of maintaining versions. On the other hand, reads are never blocked, which can be important for workloads dominated by transactions that only read values from the database.

UNIT-VI
STORAGE AND INDEXING

Data on External Storage – File Organizations and Indexing – Cluster Indexes, Primary and Secondary Indexes – Index data Structures – Hash Based Indexing – Tree base Indexing – Comparison of File Organizations – The Memory Hierarchy, RAID, Disk Space Management, Buffer Manager, Files of Records, Page Formats, record Formats.

1. Introduction:

The basic abstraction of data in a DBMS is a collection of records, or a file,and each file consists of one or more pages. The files and access methodssoftware layer organizes data carefully to support fast access to desired subsetsof records.

A file organization is a method of arranging the records in a file when thefile is stored on disk. Consider a file of employee records, each containing age, name, and salfields. If we want to retrieveemployee records in order of increasing age, sorting the file by age is a good fileorganization, but the sort order is expensive to maintain if the file is frequentlymodified.

A technique called indexing can help when we have to access a collection ofrecords in multiple ways, in addition to efficiently supporting various kinds ofselection.

2. DATA ON EXTERNAL STORAGE

DBMS stores vast quantities of data, and the data must persist across programexecutions. Therefore, data is stored on external storage devices such asdisks and tapes, and fetched into main memory as needed for processing. Theunit of information read from or written to disk is a page. The size of a pageis a DBMS parameter, and typical values are 4KB or 8KB.

The cost of page I/O (input from disk to main memory and output from memoryto disk) dominates the cost of typical database operations, and database systems are carefully optimized to minimize this cost. While the details of howfiles of records are physically stored on disk and how main memory is utilized, the following points are important to keep in mind:

• Disks are the most important external storage devices. They allow us toretrieve any page at a (more or less) fixed cost per page.

• Tapes are sequential access devices and force us to read data one page afterthe other.

• Each record in a file has a unique identifier called a record id, or rid forshort. A rid has the property that we can identify the disk address of thepage containing the record by using the rid.

Data is read into memory for processing, and written to disk for persistentstorage, by a layer of software called the buffer manager. When the files andaccess methods layer (which we often refer to as just the file layer) needs toprocess a page, it asks the buffer manager to fetch the page, specifying thepage's rid. The buffer manager fetches the page from disk if it is not alreadyin memory.Space on disk is managed by the disk space manager, according to the DBMSsoftware architecture, when the files and access methodslayer needs additional space to hold new records in a file, it asks the diskspace manager to allocate an additional disk page for the file; it also informsthe disk space manager when it no longer needs one of its disk pages. The diskspace manager keeps track of the pages in use by the file layer; if a page is freed by the file layer, the space manager tracks this, and reuses the space if the filelayer requests a new page later on.

3. FILE ORGANIZATIONS AND INDEXING

The file of records is an important abstraction in a DBMS, and is implementedby the files and access methods layer of the code. A file can be created,destroyed, and have records inserted into and deleted from it. It also supportsscan, a scan operation allows us to step through all the records in the file one at a time. A relation is typically stored as file of records.

The file layer stores the records in a file in a collection of disk pages. It keepstrack of pages allocated to each file, and as records are inserted into and deletedfrom the file, it also tracks available space within pages allocated to the file.The simplest file structure is an unordered file, or heap file. Records in aheap file are stored in random order across the pages of the file. A heap fileorganization supports retrieval of all records, or retrieval of a particular recordspecified by its rid; the file manager must keep track of the pages allocated forthe file.
An index is a data structure that organizes data records on disk to optimizecertain kinds of retrieval operations. An index allows us to efficiently retrieveall records that satisfy search conditions on the search key fields of the index.We can also create additional indexes on a given collection of data records,each with a different search key, to speed up search operations that are notefficiently supported by the file organization used to store the data records.
Consider our example of employee records. We can store the records in a fileorganized as an index on employee age; this is an alternative to sorting the fileby age. We use the term data entry to refer to the records stored in an index file. Adata entry with search key value k, denoted as k*, contains enough informationto locate (one or more) data records with search key value k. We can efficientlysearch an index to find the desired data entries, and then use these to obtaindata records (if these are distinct from data entries).
There are three main alternatives for what to store as a data entry in an index:
1. A data entry h is an actual data record (with search key value k).

2. A data entry is a (k, rid) pair, where rid is the record id of a data recordwith search key value k.

3. A data entry is a (k. rid-list) pair, where rid-list is a list of record ids ofdata records with search key value k.

Of course, if the index is used to store actual data records, Alternative (1),each entry b is a data record with search key value k. We can think of such anindex as special file organization. Such an indexed file organization canbe used instead of, for example, a sorted file or an unordered file of records.
Alternatives (2) and (3), which contain data entries that point to data records,are independent of the file organization that is used for the indexed file the file that contains the data records). Alternative (3) offers better space utilizationthan Alternative (2), but data entries are variable in length, dependingon the number of data records with a given search key value.

3.1. Clustered Indexes

When a file is organized so that the ordering of data records is the same asor close to the ordering of data entries in some index, we say that the indexis clustered; otherwise, it clustered is an un-clustered index. An index thatuses Alternative (1) is clustered, by definition. An index that uses Alternative
(2) or (3) can be a clustered index only if the data records are sorted on then search key field. Otherwise, the order of the data records is random, definedpurely by their physical order, and there is no reasonable way to arrange thedata entries in the index in the same order.

3.2. Primary and Secondary Indexes

An index on a set of fields that includes the primary key iscalled a primary index; other indexes are called secondary indexes. An index that uses Alternative (1) is called a primary index, andone that uses Alternatives (2) or (3) is called a secondary index.

Two data entries are said to be duplicates if they have the same value for thesearch key field associated with the index. A primary index is guaranteed notto contain duplicates, but an index on other (collections of) fields can containduplicates. In general, a secondary index contains duplicates. If we knowthat no duplicates exist, that is, we know that the search key contains somecandidate key, we call the index a unique index.

4. INDEX DATA STRUCTURES

One way to organize data entries is to hash data entries on the search key.Another way to organize data entries is to build a tree-like data structure thatdirects a search for data entries. We introduce these two basic approaches illthis section. We study tree-based indexing in more detail.

4.1. Hash-Based Indexing

Vie can organize records using a technique called hashing to quickly find recordsthat have a given search key value. For example, if the file of employee recordsis hashed on the name field, we can retrieve all records about Joe.
In this approach, the records in a file are grouped in buckets, where a bucketconsists of a primary page and, possibly, additional pages linked in a chain.The bucket to which a record belongs can be determined by applying a specialfunction, called a hash function, to the search key. Given a bucket number,a hash-based index structure allows us to retrieve the primary page for thebucket in one or two disk l/Os.
On inserts, the record is inserted into the appropriate bucket, with 'overflow'pages allocated as necessary. To search for a record with a given search keyvalue, we apply the hash function to identify the bucket to which such recordsbelong and look at all pages in that bucket. If we do not have the search keyvalue for the record, for example, the index is based on sal and we want records
with a given age value, we have to scan all pages in the file.
we assume that applying the hash function to (the search keyof) a record allows us to identify and retrieve the page containing the recordwith one I/O. In practice, hash-based index structures that adjust gracefullyto inserts and deletes and allow us to retrieve the page containing a record inone to two l/Os (see Chapter 11) are known.
Hash indexing is illustrated in Figure 8.2, where the data is stored in a file thatis hashed on age; the data entries in this first index file are the actual datarecords. Applying the hash function to the age field identifies the page thatthe record belongs to. The hash function h for this example is quite simple;it converts the search key value to its binary representation and uses the twoleast significant bits as the bucket identifier.
Figure 8.2 also shows an index with search key sal that contains (sal, rid) pairsas data entries. The tid (short for record id) component of a data entry in thissecond index is a pointer to a record with search key value sal (and is shownin the figure as an arrow pointing to the data record).
Using the terminology introduced in Section 8.2, Figure 8.2 illustrates Alternatives (1) and (2) for data entries. The file of employee records is hashed onage, and Alternative (1) is used for for data entries. The second index, on salalso uses hashing to locate data entries, which are now (sal, rid of employeerecord) pairs; that is, Alternative (2) is used for data entries.
Note that the search key for an index can be any sequence of one or morefields, and it need not uniquely identify records. For example, in the salaryindex, two data entries have the same search key value 6003.

4.2. Tree-Based Indexing

An alternative to hash-based indexing is to organize records using a treelikedata structure. The data entries are arranged in sorted order by searchkey value, and a hierarchical search data structure is maintained that directssearches to the correct page of data entries.
Figure shows the employee records are organized in atree-structured index with search keyage. Each node in this figure (e.g., nodeslabeled A, B, L1, L2) is a physical page, and retrieving a node involves a diskI/O.
The lowest level of the tree, called the leaf level, contains the data entries;in the example, these are employee records. To illustrate the ideas better, wehave drawn Figure as if there were additional employee records, some withage less than 22 and some with age greater than 50. Additional records

with age less than22 would appear in leaf pages to the left page L1, and records with age greaterthan 50 would appear in leaf pages to the right of page L3.
This structure allows us to efficiently locate all data entries with search keyvalues in a desired range. All searches begin at the topmost node, called theroot, and the contents of pages in non-leaf levels direct searches to the correctleaf page. Non-leaf pages contain node pointers separated by search key values.The node pointer to the left of a key value k points to a subtree that containsonly data entries less than k. The node pointer to the right of a key value kpoints to a subtree that containsonly data entries greater than or equal to k.
In our example, suppose we want to find all data entries with 24 < age < 50.Each edge from the root node to a child node in Figure 8.2 has a label thatexplains what the corresponding subtree contains. (Although the labels for theremaining edges in the figure are not shown, they should be easy to deduce.)
In our example search, we look for data entries with search key value > 24,and get directed to the middle child, node A. Again, examining the contentsof this node, we are directed to node B. Examining the contents of node B, weare directed to leaf node Ll, which contains data entries we are looking for.
Observe that leaf nodes L2 and L3 also contain data entries that satisfy oursearch criterion. To facilitate retrieval of such qualifying entries during search,all leaf pages are maintained in a doubly-linked list. Thus, we can fetch pageL2 using the 'next' pointer on page Ll, and then fetch page L3 using the 'next'pointer on L2.
Thus, the number of disk I/Os incurred during a search is equal to the lengthof a path from the root to a leaf, plus the number of leaf pages with qualifyingdata entries. The B+ tree is an index structure that ensures that all pathsfrom the root to a leaf in a given tree are of the same length, that is, the structure is always balanced in height. Finding the correct leaf page is faster,than binary search of the pages in a sorted file because each non-leaf node canaccommodate a very large number of node-pointers, and the height of the treeis rarely more than three or four in practice. The height of a balanced tree isthe length of a path from root to leaf; in Figure 8.3, the height is three. Thenumber of l/Os to retrieve a desired leaf page is four, including the root andthe leaf page. (In practice, the root is typically in the buffer pool because itis frequently accessed, and we really incur just three I/Os for a tree of heightthree.)
The average number of children for a non-leaf node is called the fan-out ofthe tree. If every non-leaf node has n children, a tree of height h has n^h leafpages. In practice, nodes do not have the same number of children, but usingthe average value F for n, we still get a good approximation to the number ofleaf pages, F h . In practice, F is at least 100, which means a tree of height fourcontains 100 million leaf pages. Thus, we can search a file with 100 million leafpages and find the page we want using four l/Os; in contrast, binary search ofthe same file would take log21OO,000,000 (over 25) l/Os.

5. COMPARISON OF FILE ORGANIZATIONS

We now compare the costs of some simple operations for several basic file organizations on a collection of employee records. We assume that the files and indexes are organized according to the composite search key (age,sal) and thatall selection operations are specified on these fields. The organizations that we consider are the following:
• File of randomly ordered employee records, or heap file.
• File of employee records sorted on (age, sal).
• Clustered B+ tree file with search key (age, sal).
• Heap file with an un-clustered B+ tree index on (age, sal).
• Heap file with an un-clustered hash index on (age, sal).
Our goal is to emphasize the importance of the choice of an appropriate fileorganization, and the above list includes the main alternatives to consider inpractice. Obviously, we can keep the records

unsorted or sort them. We canalso choose to build an index on the data file. Note that even if the data files sorted, an index whose search key differs from the sort order behaves like anindex on a heap file.
The operations we consider are these:
• **Scan:** Fetch all records in the file. The pages in the file must be fetchedfrom disk into the buffer pool. There is also a CPU overhead per recordfor locating the record on the page (in the pool).
• **Search with Equality Selection:** Fetch all records that satisfy an equalityselection; for example, "Find the employee record for the employee withage 23 and sal 50." Pages that contain qualifying records must be fetchedfrom disk, and qualifying records must be located within retrieved pages.
• **Search with Range Selection:** Fetch all records that satisfy a rangeselection; for example, "Find all employee records with age greater than35."
• **Insert a Record:** Insert a given record into the file. We must identify thepage in the file into which the new record must be inserted, fetch that pagefrom disk, modify it to include the new record, and then write back themodified page. Depending on the file organization, we may have to fetch,modify, and write back other pages as well.
• **Delete a Record:** Delete a record that is specified using its rid. We mustidentify the page that contains the record, fetch it from disk, modify it, andwrite it back. Depending on the file organization, we may have to fetch,modify, and write back other pages as well.

5.1 Cost Model

In our comparison of file organizations, and in later chapters, we use a simplecost model that allows us to estimate the cost (in terms of execution time) ofdifferent database operations. We use B to denote the number of data pageswhen records are packed onto pages with no wasted space, and R to denotethe number of records per page. The average time to read or write a diskpage is D, and the average time to process a record (e.g., to compare a fieldvalue to a selection constant) is C. In the ha.'3hed file organization, we use afunction, called a hash function, to map a record into a range of numbers; thetime required to apply the hash function to a record is H. For tree indexes, wewill use F to denote the fan-out, which typically is at least 100 as mentionedin Section 8.3.2.
Typical values today are D = 15 milliseconds, C and H = 100 nanoseconds; wetherefore expect the cost of I/O to dominate. I/O is often (even typically) thedominant component of the cost of database operations, and so considering I/Ocosts gives us a good first approximation to the true costs. Further, CPU speedsare steadily rising, whereas disk speeds are not increasing at a similar pace. (Onthe other hand, as main memory sizes increase, a much larger fraction of theneeded pages are likely to fit in memory, leading to fewer I/O requests!) We have chosen to concentrate on the I/O component of the cost model, and weassume the simple constant C for in-memory per-record processing cost. Bearthe following observations in mind:
.. Real systems must consider other aspects of cost, such as CPU costs (andnetwork transmission costs in a distributed database).

.. Even with our decision to focus on I/O costs, an accurate model would betoo complex for our purposes of conveying the essential ideas in a simpleway.

We therefore use a simplistic model in which we just count the numberof pages read from or written to disk as a measure of I/O. We ignore theimportant issue of blocked access in our analysis-typically, disk systemsallow us to read a block of contiguous pages in a single I/O request. Thecost is equal to the time required to seek the first page in the block andtransfer all pages in the block. Such blocked access can be much cheaperthan issuing one I/O request per page in the block, especially if theserequests do not follow consecutively, because we would have an additionalseek cost for each page in the block.

5.2. Heap Files

Scan: The cost is B(D +RC) because we must retrieve each of B pages takingtime D per page, and for each page, process R records taking time C per record.
Search with Equality Selection: Suppose that we know in advance thatexactly one record matches the desired equality selection, that is, the selectionis specified on a candidate key. On average, we must scan half the file, assumingthat the record exists and the distribution of values in the search field is uniform.
For each retrieved data page, we must check all records on the page to see ifit is the desired record. The cost is O.5B(D + RC). If no record satisfies theselection, however, we must scan the entire file to verify this.
If the selection is not on a candidate key field (e.g., "Find employees aged 18"),we always have to scan the entire file because records with age = 18 could bedispersed all over the file, and we have no idea how many such records exist.
Search with Range Selection: The entire file must be scanned becausequalifying records could appear anywhere in the file, and we do not know howmany qualifying records exist. The cost is B(D + RC).
Insert: We assume that records are always inserted at the end of the file. Wemust fetch the last page in the file, add the record, and write the page back.The cost is 2D +C.
Delete: We must find the record, remove the record from the page, and writethe modified page back. We assume that no attempt is made to compact thefile to reclaim the free space created by deletions, for simplicity. The cost isthe cost of searching plus C + D.

5.3. Sorted Files

Scan: The cost is B(D +RC) because all pages must be examined. Note thatthis case is no better or worse than the case of unordered files. However, theorder in which records are retrieved corresponds to the sort order, that is, allrecords in age order, and for a given age, by sal order.
Search with Equality Selection: We assume that the equality selectionmatches the sort order (age, sal). In other words, we assume that a selectioncondition is specified on at least the first field in the composite key (e.g., age =30). If not (e.g., selection sal = t50 or department = "Toy"), the sort order does not help us and the cost is identical to that for a heap file.
We can locate the first page containing the desired record or records, shouldany qualifying records exist, with a binary search in log2B steps. (This analysisassumes that the pages in the sorted file are stored sequentially, and we canretrieve the ith page on the file directly in one disk I/O.) Each step requires a disk I/O and two comparisons. Once the page is known, the first qualifyingrecord can again be located by a binary search of the page at a cost of Clog2R.
The cost is Dlo92B+Clog2R, which is a significant improvement over searchingheap files.
Search with Range Selection: Again assuming that the range selectionmatches the composite key, the first record that satisfies the selection is locatedas for search with equality. Subsequently, data pages are sequentially retrieveduntil a record is found that does not satisfy the range selection; this is similarto an equality search with many qualifying records.The cost is the cost of search plus the cost of retrieving the set of records thatsatisfy the search. The cost of the search includes the cost of fetching the firstpage containing qualifying, or matching, records. For small range selections,all qualifying records appear on this page. For larger range selections, we haveto fetch additional pagescontaining matching records.
Insert: To insert a record while preserving the sort order, we must first findthe correct position in the file, add the record, and then fetch and rewrite allsubsequent pages (because all the old records are shifted by one slot, assumingthat the file has no empty slots). On average, we can assume that the insertedrecord belongs in the middle of the file. Therefore, we must read the latter halfof the file and then write it back after adding the new record. The cost is thatof searching to find the position of the new record plus 2 . (O.5B(D + RC)),that is, search cost plus B(D + RC).
Delete: We must search for the record, remove the record from the page, andwrite the modified page back. We must also read and write all subsequentpages because all records that follow the deleted

record must be moved up tocompact the free space. 2 The cost is the same as for an insert, that is, searchcost plus B(D + RC). Given the rid of the record to delete, we can fetch thepage containing the record directly.

5.4. Clustered Files

In a clustered file, extensive empirical study has shown that pages are usuallyat about 67 percent occupancy. Thus, the Humber of physical data pages isabout 1.5B, and we use this observation in the following analysis.

Scan: The cost of a scan is 1.5B(D + RC) because all data pages must beexamined; this is similar to sorted files, with the obvious adjustment for theincreased number of data pages. Note that our cost metric does not capturepotential differences in cost due to sequential I/O. We would expect sorted filesto be superior in this regard, although a clustered file using ISAM (rather thanB+ trees) would be close.

Search with Equality Selection: We assume that the equality selectionmatches the search key (age, sal). We can locate the first page containingthe desired record or records, should any qualifying records exist, in logF1.5Bsteps, that is, by fetching all pages from the root to the appropriate leaf. In practice, the root page is likely to be in the buffer pool and we save an I/O,but we ignore this in our simplified analysis. Each step requires a disk I/Oand two comparisons. Once the page is known, the first qualifying record canagain be located by a binary search of the page at a cost of Clog2R. The costis DlogF1.5B +Clog2 R, which is a significant improvement over searching evensorted files.

Search with Range Selection: Again assuming that the range selectionmatches the composite key, the first record that satisfies the selection is locatedas it is for search with equality. Subsequently, data pages are sequentiallyretrieved (using the next and previous links at the leaf level) until a record isfound that does not satisfy the range selection; this is similar to an equalitysearch with many qualifying records.

Insert: To insert a record, we must first find the correct leaf page in the index,reading every page from root to leaf. Then, we must add the new record. Mostof the time, the leaf page has sufficient space for the new record, and all weneed to do is to write out the modified leaf page. Occasionally, the leaf is fulland we need to retrieve and modify other pages, but this is sufficiently rarethat we can ignore it in this simplified analysis. The cost is therefore the costof search plus one write, DlogF L5B + Clog2R + D.

Delete: We must search for the record, remove the record from the page,and write the modified page back. The discussion and cost analysis for insertapplies here as well.

5.5 Heap File with Un-clustered Tree Index

The number of leaf pages in an index depends on the size of a data entry.

Scan: Consider Figure 8.1, which illustrates an un-clustered index. To do a fullscan of the file of employee records, we can scan the leaf level of the index andfor each data entry, fetch the corresponding data record from the underlyingfile, obtaining data records in the sort order (age, sal).

We can read all data entries at a cost of O.15B(D + 6.7RC) l/Os. Now comesthe expensive part: We have to fetch the employee record for each data entryin the index. The cost of fetching the employee records is one I/O per record,since the index is un-clustered and each data entry on a leaf page of the indexcould point to a different page in the employee file. The cost of this step isB R(D + C), which is prohibitively high. If we want the employee recordsin sorted order, we would be better off ignoring the index and scanning theemployee file directly, and then sorting it. A simple rule of thumb is that a filecan be sorted by a two-PASS algorithm in which each pass requires reading andwriting the entire file. Thus, the I/O cost of sorting a file with B pages is 4B,which is much less than the cost of using an un-clustered index.

Search with Equality Selection: We assume that the equality selectionmatches the sort order (age, sal). We can locate the first page containing thedesired data entry or entries, should any qualifying entries exist, in lagrO.15Bsteps, that is, by fetching all pages from the root to the appropriate leaf. Eachstep requires a disk I/O and two comparisons. Once the page is known, thefirst qua1ifying data entry can again be located by a binary search of the pageat a cost of Clog2 G. 7R. The first qualifying data record can be fetched from the employee file with another I/O. The cost is DlogpO.15B + Clag26.7R + D,which is a significant improvement over searching sorted files.

Search with Range Selection: Again assuming that the range selectionmatches the composite key, the first record that satisfies the selection is locatedas it is for search with equality. Subsequently, data entries are sequentiallyretrieved (using the next and previous links at the leaf level of the index)until a data entry is found that does not satisfy the range selection. For eachqualifying data entry, we incur one I/O to fetch the corresponding employeerecords. The cost can quickly become prohibitive as the number of records thatsatisfy the range selection increases. As a rule of thumb, if 10 percent of datarecords satisfy the selection condition, we are better off retrieving all employeerecords, sorting them, and then retaining those that satisfy the selection.

Insert: We must first insert the record in the employee heap file, at a cost of2D + C. In addition, we must insert the corresponding data entry in the index.Finding the right leaf page costs Dl09pO.15B + Cl0926.7R, and writing it outafter adding the new data entry costs another D.

Delete: We need to locate the data record in the employee file and the dataentry in the index, and this search step costs Dl09FO.15B + Cl0926.7R + D.Now, we need to write out the modified pages in the index and the data file,at a cost of 2D.

5.6 Heap File With Un-clustered Hash Index

As for un-clustered tree indexes, we assume that each data entry is one tenththe size of a data record. We consider only static hashing in our analysis, andfor simplicity we a.'3sume that there are no overflow chains.In a static hashed file, pages are kept at about 80 percent occupancy (to leavespace for future insertions and minimize overflows as the file expands). This isachieved by adding a new page to a bucket when each existing page is 80 percentfull, when records are initially loaded into a hashed file structure. The numberof pages required to store data entries is therefore 1.2.5 times the number ofpages when the entries are densely packed, that is, 1.25(0.10B) = O.125B.

The number of data entries that fit on a page is 1O(O.80R) = 8R, taking intoaccount the relative size and occupancy.The dynamic variants of hashing are less susceptible to the problem of overflow chains, and havea slight.ly higher average cost per search, but are otherwise similar to the static version.

Scan: As for an un-clustered tree index, all data entries can be retrieved inexpensively,at a cost of O.125B(D + 8RC) I/Os. However, for each entry, weincur the additional cost of one I/O to fetch the corresponding data record; thecost of this step is BR(D + C). This is prohibitively expensive, and further,results are unordered. So no one ever scans a hash index.

Search with Equality Selection: This operation is supported very efficientlyfor matching selections, that is, equality conditions are specified for each fieldin the composite search key (age, sal). The cost of identifying the page thatcontains qualifying data entries is H. Assuming that this bucket consists ofjust one page (i.e., no overflow pages), retrieving it costs D. If we assume thatwe find the data entry after scanning half the records on the page, the cost ofscanning the page is O.5(8R)C = 4RC. Finally, we have to fetch the datarecord from the employee file, which is another D. The total cost is thereforeH + 2D + 4RC, which is even lower than the cost for a tree index.

Search with Range Selection: The hash structure offers no help, and theentire heap file of employee records must be scanned at a cost of B(D + RC).

Insert: We must first insert the record in the employee heap file, at a costof 2D + C. In addition, the appropriate page in the index must be located,modified to insert a new data entry, and then written back. The additionalcost is H + 2D + C.

Delete: We need to locate the data record in the employee file and the dataentry in the index; this search step costs H + 2D + 4RC. Now, we need towrite out the modified pages in the index and the data file, at a cost of 2D.

5.7. Comparison of I/O Costs

Figure 8.4 compares I/O costs for the various file organizations that we discussed.A heap file has good storage efficiency and supports fast scanning andinsertion of records. However, it is slow for searches and deletions.

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5 BD	BD	2D	Search+D
Sorted	BD	D $\log_2 B$	Dlog2B+# matching pages	Search+BD	Search+BD
Clustered	1.5 BD	D log F1.5B	DlogF1.5B+# matching pages	Search+D	Search+D
Un-Clustered Tree Index	BD(R+0.15)	D(1+logF0.15B)	D(logF0.15B+# matching records)	D(3+logF0.15B)	Search+2D
Un-Clustered Hash Index	BD(R+0.125)	2D	BD	4D	Search+2D

A sorted file also offers good storage efficiency. but insertion and deletion ofrecords is slow. Searches are faster than in heap files. It is worth noting that,in a real DBMS, a file is almost never kept fully sorted.

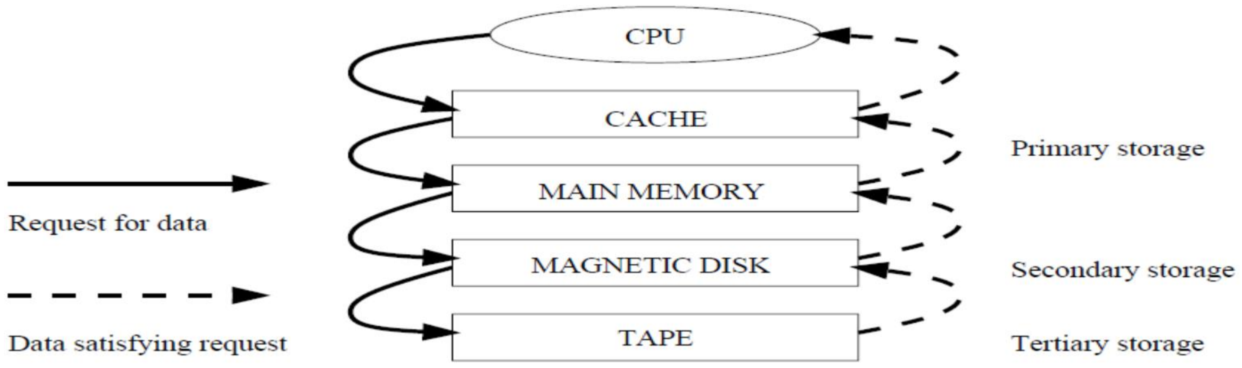
A clustered file offers all the advantages of a sorted file and supports insertsand deletes efficiently. (There is a space overhead for these benefits, relative toa sorted file, but the trade-off is well worth it.) Searches are even faster than insorted files, although a sorted file can be faster when a large number of recordsare retrieved sequentially, because of blocked I/O efficiencies.

Un-clustered tree and hash indexes offer fast searches, insertion, and deletion,but scans and range searches with many matches are slow. Hash indexes are alittle faster on equality searches, but they do not support range searches.

6. THE MEMORY HIERARCHY

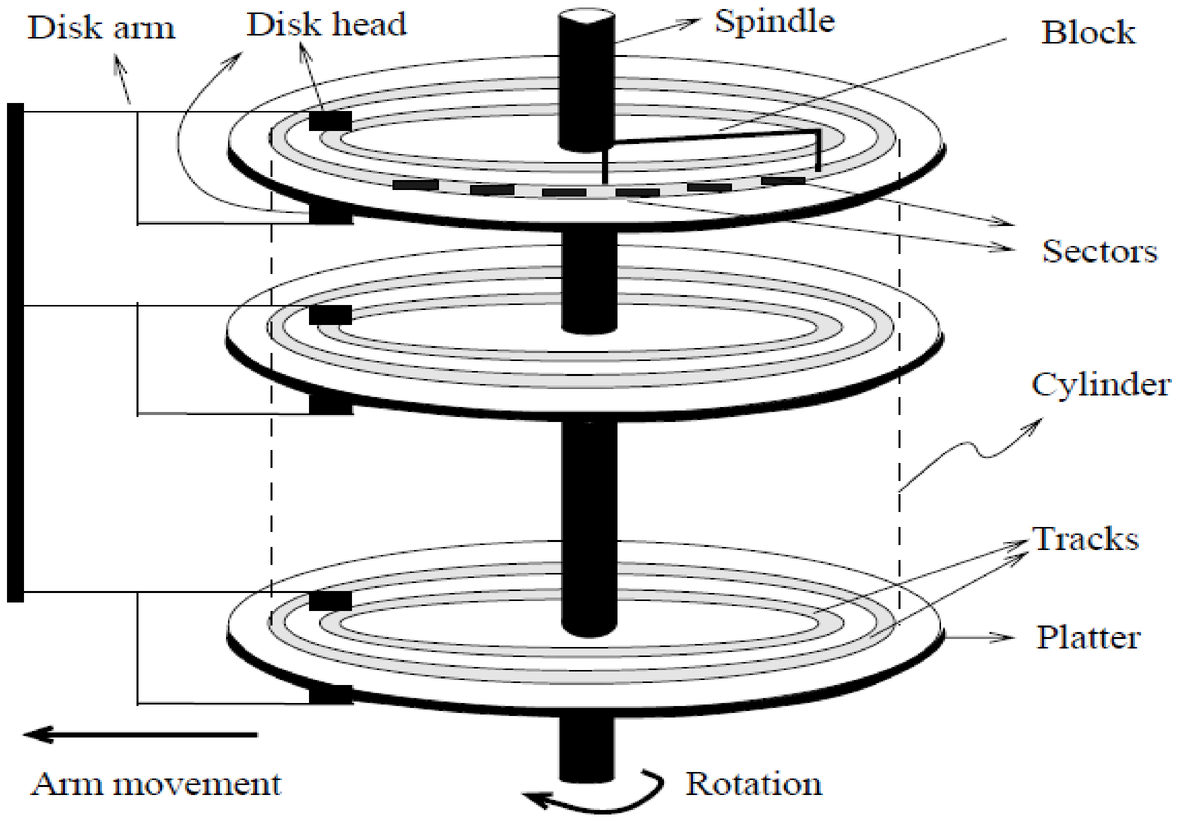
Memory in a computer system is arranged in a hierarchy, as shown in Figure. Atthe top, we have primary storage, which consists of cache and main memory, andprovides very fast access to data. Then comes secondary storage, which consists ofslower devices such as magnetic disks. Tertiary storage is the slowest class of storagedevices; for example, optical disks and tapes.

The main drawback of tapes is that they are sequential access devices. We mustessentially step through all the data in order and cannot directly access a given locationon tape. For example, to access the last byte on a tape, we would have to windthrough the entire tape first. This makes tapes unsuitable for storing operational data,or data that is frequently accessed. Tapes are mostly used to back up operational dataperiodically.



6.1. Magnetic Disks

Magnetic disks support direct access to a desired location and are widely used fordatabase applications. A DBMS provides seamless access to data on disk; applicationsneed not worry about whether data is in main memory or disk. To understand howdisks work, consider Figure 7.2, which shows the structure of a disk in simplified form.Data is stored on disk in units called disk blocks. A disk block is a contiguoussequence of bytes and is the unit in which data is written to a disk and read from adisk. Blocks are arranged in concentric rings called tracks, on one or more platters.
Tracks can be recorded on one or both surfaces of a platter; we refer to platters assingle-sided or double-sided accordingly. The set of all tracks with the same diameter iscalled a cylinder, because the space occupied by these tracks is shaped like a cylinder;a cylinder contains one track per platter surface. Each track is divided into arcs calledsectors, whose size is a characteristic of the disk and cannot be changed. The size ofa disk block can be set when the disk is initialized as a multiple of the sector size.



An array of disk heads, one per recorded surface, is moved as a unit; when one headis positioned over a block, the other heads are in identical positions with respect totheir platters. To read or write a block, a disk head must be positioned on top of theblock. As the size of a platter decreases, seek times also decrease since we have tomove a disk head a smaller distance. Typical platter diameters are 3.5 inches and 5.25inches.Current systems typically allow at most one disk head to read or write at any one time.All the disk heads cannot read or write in parallel, this technique would increase datatransfer rates by a factor equal to the number of disk heads, and considerably speedup sequential scans. The reason they cannot is that it is very difficult to ensure thatall the heads are perfectly aligned on the corresponding tracks. Current approachesare both expensive and more prone to faults as compared to disks with a single activehead. In practice very few commercial products support this capability, and then onlyin a limited way; for example, two disk heads may be able to operate in parallel.

A disk controller interfaces a disk drive to the computer. It implements commandsto read or write a sector by moving the arm assembly and transferring data to andfrom the disk surfaces. A checksum is computed for when data is written to a sectorand stored with the sector. The checksum is computed again when the data on thesector is read back. If the sector is corrupted or the read is faulty for some reason,it is very unlikely that the checksum computed when the sector is read matches thechecksum computed when the sector was written. The controller computes checksumsand if it detects an error, it tries to read the sector again. (Of course, it signals afailure if the sector is corrupted and read fails repeatedly.)

While direct access to any desired location in main memory takes approximately thesame time, determining the time to access a location on disk is more complicated. Thetime to access a disk block has several components. Seek time is the time taken tomove the disk heads to the track on which a desired block is located. Rotationaldelay is the waiting time for the desired block to rotate under the disk head; it isthe time required for half a rotation on average and is usually less than seek time.Transfer time is the time to actually read or write the data in the block once thehead is positioned, that is, the time for the disk to rotate over the block.

6.2. Performance Implications of Disk Structure

1. Data must be in memory for the DBMS to operate on it.
2. The unit for data transfer between disk and main memory is a block; if a singleitem on a block is needed, the entire block is transferred. Reading or writing adisk block is called an I/O (for input/output) operation.
3. The time to read or write a block varies, depending on the location of the data:access time = seek time + rotational delay + transfer time

These observations imply that the time taken for database operations is affected significantly by how data is stored on disks. The time for moving blocks to or from diskusually dominates the time taken for database operations. To minimize this time, itis necessary to locate data records strategically on disk, because of the geometry andmechanics of disks. In essence, if two records are frequently used together, we shouldplace them close together. The `closest' that two records can be on a disk is to be onthe same block. In decreasing order of closeness, they could be on the same track, thesame cylinder, or an adjacent cylinder.

Two records on the same block are obviously as close together as possible, because theyare read or written as part of the same block. As the platter spins, other blocks onthe track being read or written rotate under the active head. In current disk designs,all the data on a track can be read or written in one revolution. After a track is reador written, another disk head becomes active, and another track in the same cylinderis read or written. This process continues until all tracks in the current cylinder areread or written, and then the arm assembly moves (in or out) to an adjacent cylinder.

Thus, we have a natural notion of `closeness' for blocks, which we can extend to anotion of next and previous blocks.Exploiting this notion of next by arranging records so that they are read or written sequentially is very important for reducing the time spent in disk I/Os. Sequentialaccess minimizes seek time and rotational delay and is much faster than random access.

7. RAID

Disks are potential bottlenecks for system performance and storage system reliability.Even though disk performance has been improving continuously, microprocessor performancehas advanced much more rapidly. The performance of microprocessors hasimproved at about 50 percent or more per year, but disk access times have improvedat a rate of about 10 percent per year and disk transfer rates at a rate of about 20percent per year. In addition, since disks contain mechanical elements, they have muchhigher failure rates than electronic parts of a computer system. If a disk fails, all the
data stored on it is lost.
A disk array is an arrangement of several disks, organized so as to increase performanceand improve reliability of the resulting storage system. Performance is increasedthrough data striping. Data striping distributes data over several disks to give theimpression of having a single large, very fast disk. Reliability is improved throughredundancy. Instead of having a single copy of the data, redundant information ismaintained. The redundant information is carefully organized so that in case of adisk failure, it can be used to reconstruct the contents of the failed disk. Disk arraysthat implement a combination of data striping and redundancy are called redundantarrays of independent disks, or in short, RAID.1 Several RAID organizations, referredto as RAID levels, have been proposed. Each RAID level represents a differenttrade-off between reliability and performance.In the remainder of this section, we will first discuss data striping and redundancy andthen introduce the RAID levels that have become industry standards.

7.1. Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that storethe data requested. These disk blocks may reside on a single disk in the array or maybe distributed over several disks in the array. Then the set of blocks is retrieved fromthe disk(s) involved. Thus, how we distribute the data over the disks in the arrayinfluences how many disks are involved when an I/O request is processed.
In data striping, the data is segmented into equal-size partitions that are distributedover multiple disks. The size of a partition is called the striping unit. The partitionsare usually distributed using a round robin algorithm: If the disk array consists of Ddisks, then partition i is written onto disk i mod D.

7.2. Redundancy

While having more disks increases storage system performance, it also lowers overallstorage system reliability. Assume that the mean-time-to-failure, or MTTF, ofa single disk is 50; 000 hours (about 5:7 years). Then, the MTTF of an array of100 disks is only 50; 000=100 = 500 hours or about 21 days, assuming that failuresoccur independently and that the failure probability of a disk does not change overtime. (Actually, disks have a higher failure probability early and late in their lifetimes.
Early failures are often due to undetected manufacturing defects; late failures occurReliability of a disk array can be increased by storing redundant information. If adisk failure occurs, the redundant information is used to reconstruct the data on thefailed disk. Redundancy can immensely increase the MTTF of a disk array. Whenincorporating redundancy into a disk array design, we have to make two choices. First,we have to decide where to store the redundant information. We can either store theredundant information on a small number of check disks or we can distribute theredundant information uniformly over all disks.The second choice we have to make is how to compute the redundant information.Most disk arrays store parity information: In the parity scheme, an extra check diskcontains information that can be used to recover from failure of any one disk in thearray. Assume that we have a disk array with D disks and consider the first bit oneach data disk. Suppose that i of the D data bits are one. The first bit on the checkdisk is set to one if i is odd, otherwise it is

set to zero. This bit on the check disk iscalled the parity of the data bits. The check disk contains parity information for eachset of corresponding D data bits.
In a RAID system, the disk array is partitioned into reliability groups, where areliability group consists of a set of data disks and a set of check disks. A commonredundancy scheme (see box) is applied to each group. The number of check disksdepends on the RAID level chosen. In the remainder of this section, we assume forease of explanation that there is only one reliability group. The reader should keepin mind that actual RAID implementations consist of several reliability groups, andthat the number of groups plays a role in the overall reliability of the resulting storage system.

7.3. Levels of Redundancy

Throughout the discussion of the different RAID levels, we consider sample data that would just fit on four disks. That is, without any RAID technology our storage systemwould consist of exactly four data disks. Depending on the RAID level chosen, thenumber of additional disks varies from zero to four.

Level 0: Nonredundant

A RAID Level 0 system uses data striping to increase the maximum bandwidth available.No redundant information is maintained. While being the solution with thelowest cost, reliability is a problem, since the MTTF decreases linearly with the numberof disk drives in the array. RAID Level 0 has the best write performance of allRAID levels, because absence of redundant information implies that no redundant informationneeds to be updated!
In our example, the RAID Level 0 solution consists of only four data disks. Independentof the number of data disks, the effective space utilization for a RAID Level 0 systemis always 100 percent.

Level 1: Mirrored

A RAID Level 1 system is the most expensive solution. Instead of having one copy ofthe data, two identical copies of the data on two different disks are maintained. Thistype of redundancy is often called mirroring. Every write of a disk block involves awrite on both disks. These writes may not be performed simultaneously, since a globalsystem failure (e.g., due to a power outage) could occur while writing the blocks andthen leave both copies in an inconsistent state.

Level 0+1: Striping and Mirroring

RAID Level 0+1|sometimes also referred to as RAID level 10 combines striping andmirroring. Thus, as in RAID Level 1, read requests of the size of a disk block can bescheduled both to a disk or its mirror image. In addition, read requests of the size ofseveral contiguous blocks benefit from the aggregated bandwidth of all disks. The costfor writes is analogous to RAID Level 1.
As in RAID Level 1, our example with four data disks requires four check disks and the effective space utilization is always 50 percent.

Level 2: Error-Correcting Codes

In RAID Level 2 the striping unit is a single bit. The redundancy scheme used isHamming code. In our example with four data disks, only three check disks are needed.In general, the number of check disks grows logarithmically with the number of datadisks.
Striping at the bit level has the implication that in a disk array with D data disks,the smallest unit of transfer for a read is a set of D blocks. Thus, Level 2 is good forworkloads with many large requests since for each request the aggregated bandwidthof all data disks is used. But RAID Level 2 is bad for small requests of the size ofan individual block for the same reason.

Level 3: Bit-Interleaved Parity

While the redundancy schema used in RAID Level 2 improves in terms of cost uponRAID Level 1, it keeps more redundant information than is necessary. Hamming code,as used in RAID Level 2, has the advantage of being able to identify which disk hasfailed. But disk controllers can easily detect which disk has failed. Thus, the checkdisks do not need to contain information to identify the failed disk. Information to recover the lost data is sufficient. Instead of using several disks to store Hamming code,RAID Level 3 has a single check disk with parity information. Thus, the reliability overhead for RAID Level 3 is a single disk, the lowest overhead possible.The performance characteristics of RAID Level 2 and RAID Level 3 are very similar.RAID Level 3 can also process only one I/O at a time, the minimum transfer unit isD blocks, and a write requires a read-modify-write cycle.

Level 4: Block-Interleaved Parity

RAID Level 4 has a striping unit of a disk block, instead of a single bit as in RAIDLevel 3. Block-level striping has the advantage that read requests of the size of a diskblock can be served entirely by the disk where the requested block resides. Large readrequests of several disk blocks can still utilize the aggregated bandwidth of the D disks.The write of a single block still requires a read-modify-write cycle, but only one datadisk and the check disk are involved. The parity on the check disk can be updatedwithout reading all D disk blocks, because the new parity can be obtained by noticing the differences between the old data block and the new data block and then applyingthe difference to the parity block on the check disk:

NewParity = (OldData XOR NewData) XOR OldParity

The read-modify-write cycle involves reading of the old data block and the old parityblock, modifying the two blocks, and writing them back to disk, resulting in four diskaccesses per write. Since the check disk is involved in each write, it can easily becomethe bottleneck.

Level 5: Block-Interleaved Distributed Parity

RAID Level 5 improves upon Level 4 by distributing the parity blocks uniformly overall disks, instead of storing them on a single check disk. This distribution has twoadvantages. First, several write requests can potentially be processed in parallel, sincethe bottleneck of a unique check disk has been eliminated. Second, read requests havea higher level of parallelism. Since the data is distributed over all disks, read requestsinvolve all disks, whereas in systems with a dedicated check disk the check disk neverparticipates in reads.

A RAID Level 5 system has the best performance of all RAID levels with redundancyfor small and large read and large write requests. Small writes still require a read-modify-write cycle and are thus less efficient than in RAID Level 1.

Level 6: P+Q Redundancy

The motivation for RAID Level 6 is the observation that recovery from failure of asingle disk is not su_cient in very large disk arrays. First, in large disk arrays, asecond disk might fail before replacement of an already failed disk could take place.In addition, the probability of a disk failure during recovery of a failed disk is notnegligible.

A RAID Level 6 system uses Reed-Solomon codes to be able to recover from up to twosimultaneous disk failures. RAID Level 6 requires (conceptually) two check disks, butit also uniformly distributes redundant information at the block level as in RAID Level5. Thus, the performance characteristics for small and large read requests and for largewrite requests are analogous to RAID Level 5. For small writes, the read-modify-writeprocedure involves six instead of four disks as compared to RAID Level 5, since twoblocks with redundant information need to be updated.

Choice of RAID Levels

If data loss is not an issue, RAID Level 0 improves overall system performance atthe lowest cost. RAID Level 0+1 is superior to RAID Level 1. The main applicationareas for RAID Level 0+1 systems are small storage subsystems where the cost ofmirroring is moderate. Sometimes RAID Level 0+1 is used for applications that havea high percentage of writes in their workload, since RAID Level 0+1 provides the bestwrite performance. RAID levels 2 and 4 are always inferior to RAID levels 3 and 5,respectively. RAID Level 3 is appropriate for workloads consisting mainly of largetransfer requests of several contiguous blocks. The performance of a RAID Level 3system is bad for workloads with many small requests of a single disk block. RAIDLevel 5 is a good general-purpose solution. It provides high performance for largerequests as well as for small requests. RAID Level 6 is appropriate if a higher level ofreliability is required.

8. DISK SPACE MANAGEMENT

The lowest level of software in the DBMS architecture discussed in last chapter, calledthe disk space manager, manages space on disk. Abstractly, the disk space managersupports the concept of a page as a unit of data, and provides commands to allocateor de-allocate a page and read or write a page. The size of a page is chosen to be thesize of a disk block and pages are stored as disk blocks so that reading or writing apage can be done in one disk I/O.

It is often useful to allocate a sequence of pages as a contiguous sequence of blocks tohold data that is frequently accessed in sequential order. This capability is essentialfor exploiting the advantages of sequentially accessing disk blocks, which we discussedearlier in this chapter. Such a capability, if desired, must be provided by the disk spacemanager to higher-level layers of the DBMS.Thus, the disk space manager hides details of the underlying hardware (and possiblythe operating system) and allows higher levels of the software to think of the data asa collection of pages.

8.1. Keeping Track of Free Blocks

A database grows and shrinks as records are inserted and deleted over time. Thedisk space manager keeps track of which disk blocks are in use, in addition to keepingtrack of which pages are on which disk blocks. Although it is likely that blocks areinitially allocated sequentially on disk, subsequent allocations and de-allocations couldin general create `holes.'

One way to keep track of block usage is to maintain a list of free blocks. As blocks arede-allocated, we canadd them to the free list for future use. A pointer to the first block on the free blocklist is stored in a known location on disk.A second way is to maintain a bitmap with one bit for each disk block, which indicateswhether a block is in use or not. A bitmap also allows very fast identification andallocation of contiguous areas on disk. This is difficult to accomplish with a linked listapproach.

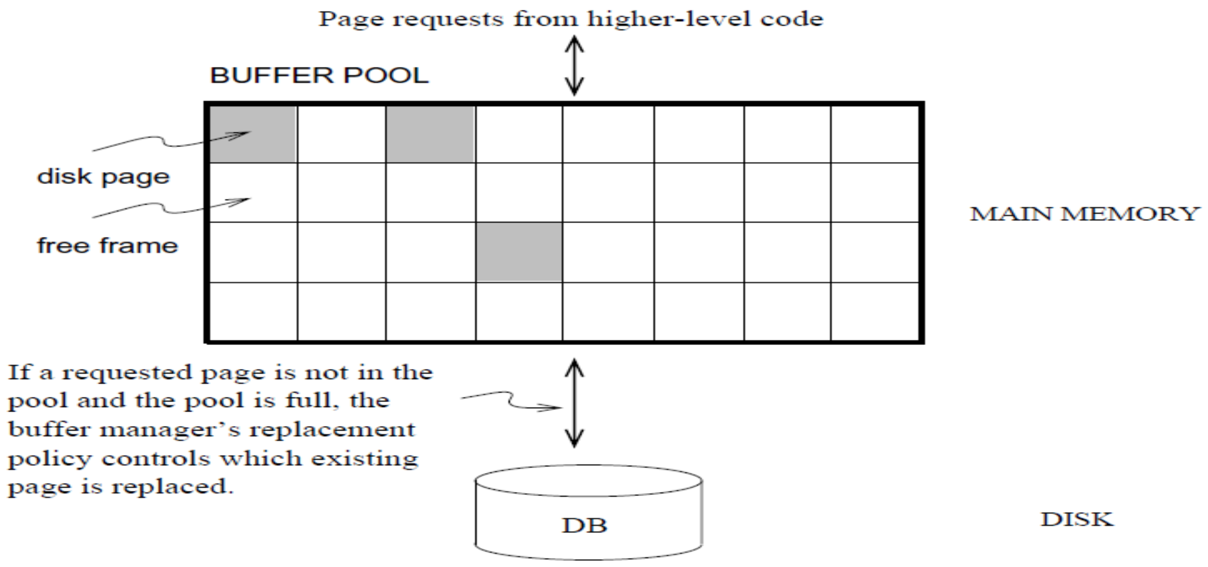
8.2. Using OS File Systems to Manage Disk Space

Operating systems also manage space on disk. Typically, an operating system supportsthe abstraction of a file as a sequence of bytes. The OS manages space on the diskand translates requests such as Read byte i of file f" into corresponding low-level instructions: “Read block m of track t of cylinder c of disk d." A database disk space
manager could be built using OS files. For example, the entire database could residein one or more OS files for which a number of blocks are allocated (by the OS) andinitialized. The disk space manager is then responsible for managing the space in these OS files.Many database systems do not rely on the OS file system and instead do their owndisk management, either from scratch or by extending OS facilities. The reasonsare practical as well as technical. One practical reason is that a DBMS vendor whowishes to support several OS platforms cannot assume features specific to any OS,
for portability, and would therefore try to make the DBMS code as self-contained aspossible. A technical reason is that on a 32-bit system, the largest file size is 4 GB,whereas a DBMS may want to access a single file larger than that. A related problem isthat typical OS files cannot span disk devices, which is often desirable or even necessaryin a DBMS.

9. BUFFER MANAGER

To understand the role of the buffer manager, consider a simple example. Supposethat the database contains 1,000,000 pages, but only 1,000 pages of main memory areavailable for holding data. Consider a query that requires a scan of the entire file.Because all the data cannot be brought into main memory at one time, the DBMSmust bring pages into main memory as they are needed and, in the process, decidewhat existing page in main memory to replace to make space for the new page. Thepolicy used to decide which page to replace is called the replacement policy.

In terms of the DBMS architecture presented in the last chapter, the buffer manager isthe softwarelayer that is responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning itinto a collection of pages, which we collectively refer to as the buffer pool. The mainmemory pages in the buffer pool are called frames; it is convenient to think of themas slots that can hold a page.


Page requests from higher-level code
BUFFER POOL
disk page
free frame
MAIN MEMORY
If a requested page is not in the pool and the pool is full, the buffer manager's replacement policy controls which existing page is replaced.
DB
DISK

Buffer Pool

Higher levels of the DBMS code can be written without worrying about whether datapages are in memory or not; they ask the buffer manager for the page, and it is broughtinto a frame in the buffer pool if it is not already there. Of course, the higher-levelcode that requests a page must also release the page when it is no longer needed, byinforming the buffer manager, so that the frame containing the page can be reused.The higher-level code must also inform the buffer manager if it modifies the requestedpage; the buffer manager then makes sure that the change is propagated to the copy of the page on disk to the buffer pool itself, the buffer manager maintains some bookkeepinginformation, and two variables for each frame in the pool: pin count and dirty. Thenumber of times that the page currently in a given frame has been requested butnot released, the number of current users of the page is recorded in the pin countvariable for that frame. The Boolean variable dirty indicates whether the page hasbeen modified since it was brought into the buffer pool from disk.

Initially, the pin count for every frame is set to 0, and the dirty bits are turned off.When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page, and if soincrements the pin_count of that frame. If the page is not in the pool, the buffermanager brings it in as follows:

(a) Chooses a frame for replacement, using the replacement policy, and increments its pin count.

(b) If the dirty bit for the replacement frame is on, writes the page it containsto disk (that is, the disk copy of the page is overwritten with the contents ofthe frame).

(c) Reads the requested page into the replacement frame.

2. Returns the (main memory) address of the frame containing the requested pageto the requestor.

Incrementing pin count is often called pinning the requested page in its frame. Whenthe code that calls the buffer manager and requests the page subsequently calls thebuffer manager and releases the page, the pin count of the frame containing the requestedpage is decremented. This is called unpinning the page. If the requestor hasmodified the page, it also informs the buffer manager of this at the time that it unpinsthe page, and the dirty bit for the frame is set. The buffer manager will not

readanother page into a frame until its pin count becomes 0, that is, until all requestors ofthe page have unpinned it.
If a requested page is not in the buffer pool, and if a free frame is not available in thebuffer pool, a frame with pin count 0 is chosen for replacement. If there are many suchframes, a frame is chosen according to the buffer manager's replacement policy.When a page is eventually chosen for replacement, if the dirty bit is not set, it meansthat the page has not been modified since being brought into main memory. Thus,there is no need to write the page back to disk; the copy on disk is identical to the copyin the frame, and the frame can simply be overwritten by the newly requested page.
Otherwise, the modifications to the page must be propagated to the copy on disk.

9.1. Buffer Replacement Policies

The policy that is used to choose an unpinned page for replacement can affect the timetaken for database operations considerably. Many alternative policies exist, and eachis suitable in different situations.The best-known replacement policy is least recently used (LRU). This can be implemented in the buffer manager using a queue of pointers to frames with pin count 0.
A frame is added to the end of the queue when it becomes a candidate for replacement(that is, when the pin count goes to 0). The page chosen for replacement is the one inthe frame at the head of the queue.A variant of LRU, called clock replacement, has similar behavior but less overhead.The idea is to choose a page for replacement using a current variable that takes onvalues 1 through N, where N is the number of buffer frames, in circular order. Wecan think of the frames being arranged in a circle, like a clock's face, and current as aclock hand moving across the face. In order to approximate LRU behavior, each framealso has an associated referenced bit, which is turned on when the page pin count goesto 0.
The current frame is considered for replacement. If the frame is not chosen for replacement,current is incremented and the next frame is considered; this process is repeateduntil some frame is chosen. If the current frame has pin count greater than 0, then itis not a candidate for replacement and current is incremented. If the current framehas the referenced bit turned on, the clock algorithm turns the referenced bit off andincrements current, this way, a recently referenced page is less likely to be replaced.If the current frame has pin count 0 and its referenced bit is off, then the page in it ischosen for replacement. If all frames are pinned in some sweep of the clock hand (thatis, the value of current is incremented until it repeats), this means that no page in thebuffer pool is a replacement candidate.The LRU and clock policies are not always the best replacement strategies for adatabase system, particularly if many user requests require sequential scans of thedata. Consider the following illustrative situation. Suppose the buffer pool has 10frames, and the file to be scanned has 10 or fewer pages. Assuming, for simplicity,that there are no competing requests for pages, only the first scan of the file does anyI/O. Page requests in subsequent scans will always find the desired page in the bufferpool. On the other hand, suppose that the file to be scanned has 11 pages (which isone more than the number of available pages in the buffer pool). Using LRU, everyscan of the file will result in reading every page of the file. In this situation, calledsequential flooding, LRU is the worst possible replacement strategy.and random, among others.
The details of these policies should be evident from their names and the precedingdiscussion of LRU and clock.

9.2. Buffer Management in DBMS versus OS

Virtual memory in OS and Buffer Manager in DBMS are same, in two cases the goal is to provide access to more data than will fit in main memory, and the basic idea is to bring inpages from disk to main memory as needed, replacing pages that are no longer neededin main memory. Why can't we build a DBMS using the virtual memory capability ofan OS? A DBMS can often predict the order in which pages will be accessed, or pagereference patterns, much more accurately than is typical in an

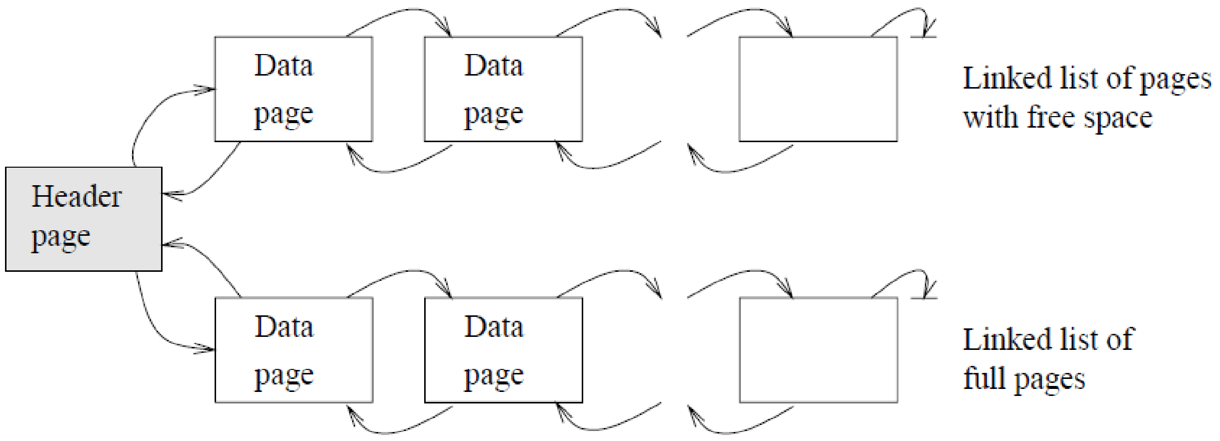
OS environment, andit is desirable to utilize this property. Further, a DBMS needs more control over whena page is written to disk than an OS typically provides.
A DBMS can often predict reference patterns because most page references are generated by higher-level operations (such as sequential scans or particular implementationsof various relational algebra

10. FILES OF RECORDS

We now turn our attention from the way pages are stored on disk and brought intomain memory to the way pages are used to store records and organized into logicalcollections or files. Higher levels of the DBMS code treat a page as effectively beinga collection of records, ignoring the representation and storage details. In fact, theconcept of a collection of records is not limited to the contents of a single page; a fileof records is a collection of records that may reside on several pages. In this section, we consider how a collection of pages can be organized as a file.
Each record has a unique identifier called a record id, or rid for short. The basic file structure that we consider, called a heap file, stores records in randomorder and supports retrieval of all records or retrieval of a particular record specifiedby its rid. Sometimes we want to retrieve records by specifying some condition onthe fields of desired records, for example, “Find all employee records with age 35." Tospeed up such selections, we can build auxiliary data structures that allow us to quicklyfind the rid’s of employee records that satisfy the given selection condition. Such anauxiliary structure is called an index;

10.1. Heap Files

The simplest file structure is an unordered file or heap file. The data in the pages ofa heap file is not ordered in any way, and the only guarantee is that one can retrieveall records in the file by repeated requests for the next record. Every record in the filehas a unique rid, and every page in a file is of the same size.


Data page
Data page
Linked list of pages with free space
Header page
Data page
Data page
Linked list of full pages

Heap File Organization with Linked List

Supported operations on a heap file include create and destroy files, insert a record,delete a record with a given rid, get a record with a given rid, and scan all records inthe file. To get or delete a record with a given rid, note that we must be able to findthe id of the page containing the record, given the id of the record.
We discuss two alternative ways to maintain this information. In eachof these alternatives, pages must hold two pointers (which are page ids) for file-levelbookkeeping in addition to the data.

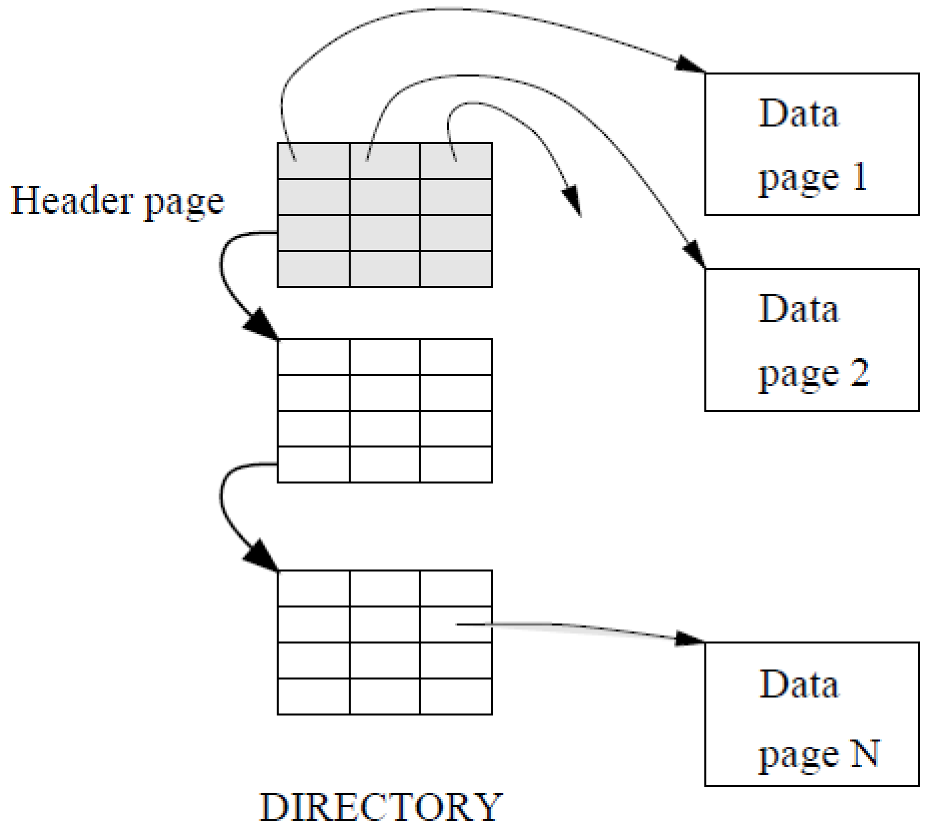
10.1.1. Linked List of Pages

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMScan remember where the first page is located by maintaining a table containing pairsof hheap file name; page 1 address i in a known location on disk. We call the first pageof the file the header page.

An important task is to maintain information about empty slots created by deleting arecord from the heap file. This task has two distinct parts: how to keep track of freespace within a page and how to keep track of pages that have some free space.The second part can be addressed by maintaininga doubly linked list of pages with free space and a doubly linked list of full pages;together, these lists contain all pages in the heap file.

10.1.2. Directory of Pages

An alternative to a linked list of pages is to maintain a directory of pages. TheDBMS must remember where the first directory page of each heap file is located. Thedirectory is itself a collection of pages and is shown as a linked list in Figure 7.5.



Heap File organization with directory

Each directory entry identifies a page (or a sequence of pages) in the heap file. As theheap file grows or shrinks, the number of entries in the directory, and possibly thenumber of pages in the directory itself, grows or shrinks correspondingly. Note thatsince each directory entry is quite small in comparison to a typical page, the size ofthe directory is likely to be very small in comparison to the size of the heap file.

11. Page Formats:

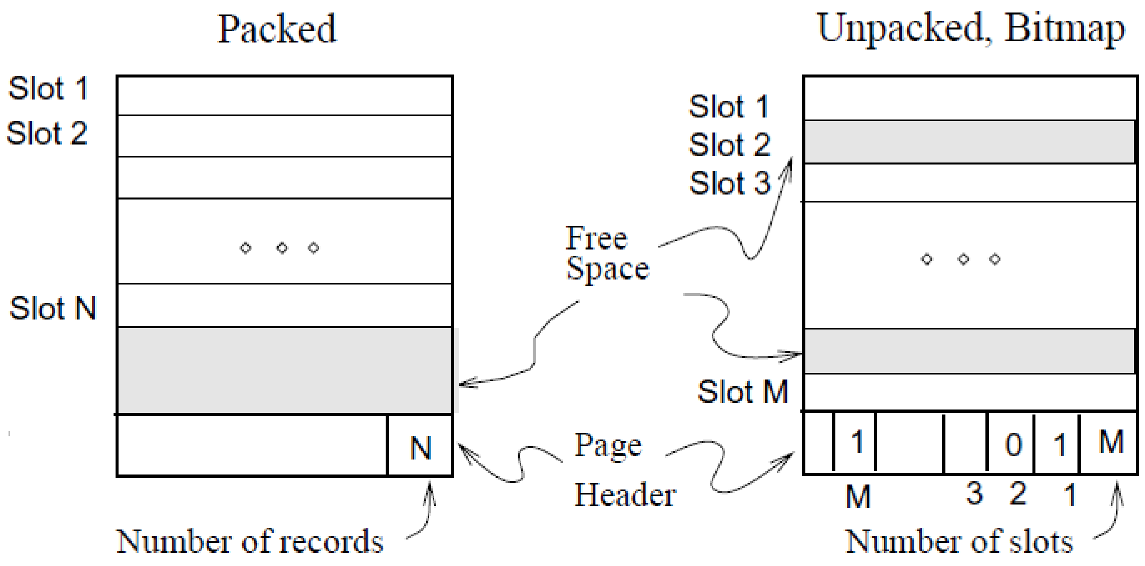
Free space can be managed by maintaining a bit per entry, indicating whether thecorresponding page has any free space, or a count per entry, indicating the amount offree space on the page. If the file contains variable-length records, we can examine thefree space count for an entry to determine if the record will _t on the page pointed toby the entry. Since several entries fit on a directory page, we can efficiently search fora data page with enough space to hold a record that is to be inserted.

I/O issues, but higher levelsof the DBMS see data as a collection of records. In this section, we consider how acollection of records can be arranged on a page. We can think of a page as a collection

of slots, each of which contains a record. A record is identified by using the pairhpage id; slot numberi; this is the record id (rid). (We remark that an alternative wayto identify records is to assign each record a unique integer as its rid and to maintaina table that lists the page and slot of the corresponding record for each rid. Due tothe overhead of maintaining this table, the approach of using hpage id; slot numberias an rid is more common.)
We now consider some alternative approaches to managing slots on a page. The mainconsiderations are how these approaches support operations such as searching, inserting, or deleting records on a page.

11.1. Fixed-Length Records

If all records on the page are guaranteed to be of the same length, record slots areuniform and can be arranged consecutively within a page. At any instant, some slotsare occupied by records, and others are unoccupied. When a record is inserted intothe page, we must locate an empty slot and place the record there. The main issuesare how we keep track of empty slots and how we locate all records on a page. Thealternatives hinge on how we handle the deletion of a record.



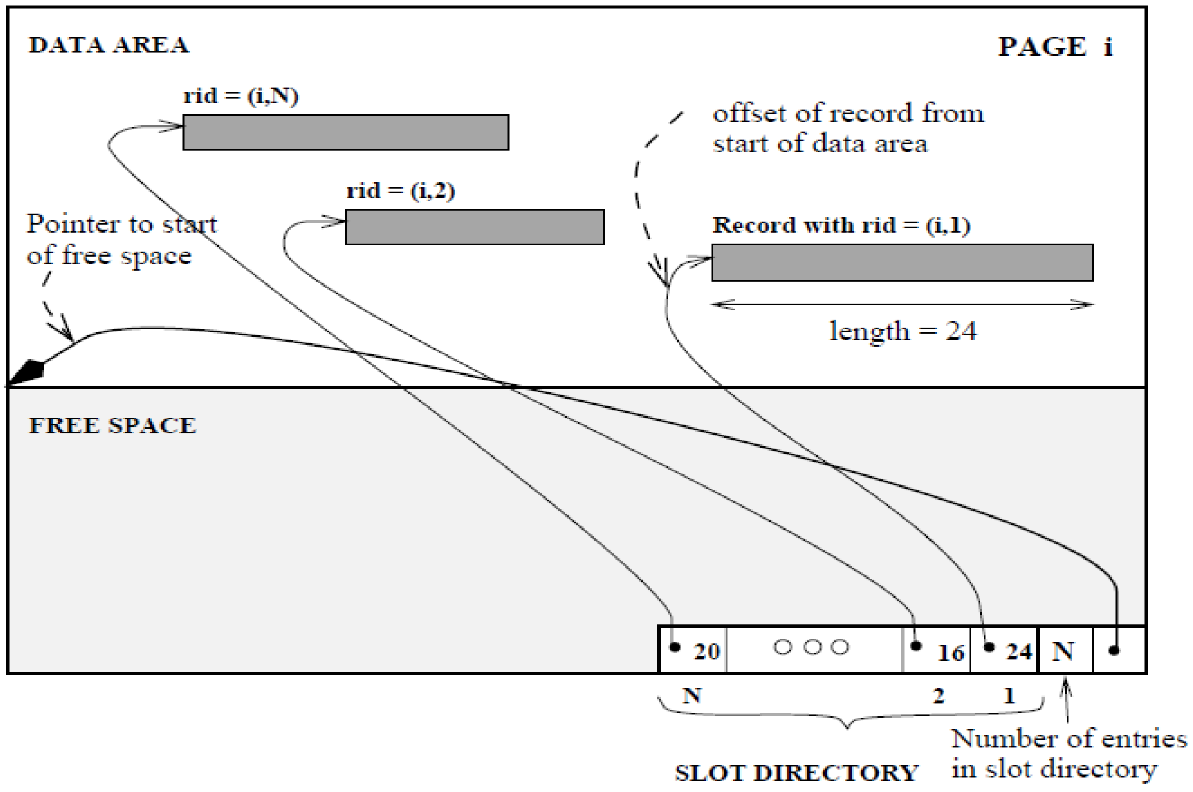
Alternative Page Organization for Fixed Length Records

The first alternative is to store records in the first N slots (where N is the numberof records on the page); whenever a record is deleted, we move the last record on thepage into the vacated slot. This format allows us to locate the ith record on a page bya simple offset calculation, and all empty slots appear together at the end of the page.However, this approach does not work if there are external references to the recordthat is moved (because the rid contains the slot number, which is now changed).The second alternative is to handle deletions by using an array of bits, one per slot,to keep track of free slot information. Locating records on the page requires scanningthe bit array to find slots whose bit is on; when a record is deleted, its bit is turnedoff. The two alternatives for storing fixed-length records are illustrated in Figure.Note that in addition to the information about records on the page, a page usuallycontains additional file-level information (e.g., the id of the next page in the file). Thefigure does not show this additional information.The slotted page organization described for variable-length records inPrevious Section canalso be used for fixed-length records. It becomes attractive if we need to move recordsaround on a page for reasons other than keeping track of space freed by deletions.

11.2. Variable-Length Records

If records are of variable length, then we cannot divide the page into a fixed collectionof slots. The problem is that when a new record is to be inserted, we have to find anempty slot of just the right

length, if we use a slot that is too big, we waste space,and obviously we cannot use a slot that is smaller than the record length. Therefore,when a record is inserted, we must allocate just the right amount of space for it, andwhen a record is deleted, we must move records to the hole created by the deletion,in order to ensure that all the free space on the page is contiguous. Thus, the ability
to move records on a page becomes very important.


DATA AREA
PAGE i
rid = (i,N)
offset of record from start of data area
rid = (i,2)
Pointer to start of free space
Record with rid = (i,1)
length = 24
FREE SPACE
20
16
24
N
N
2
1
SLOT DIRECTORY
Number of entries in slot directory

Page Organization for Variable Length Records

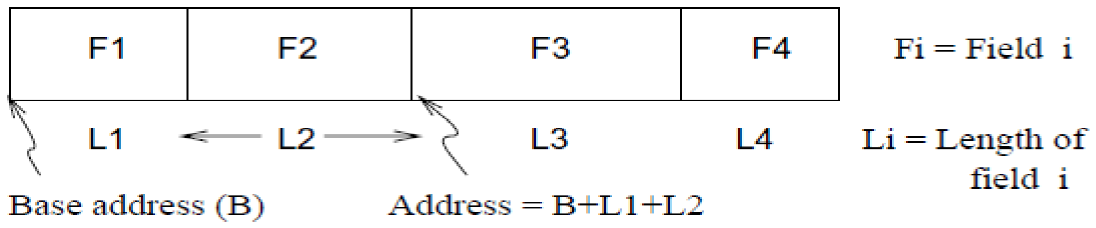
The most flexible organization for variable-length records is to maintain a directoryof slots for each page, with a hrecord offset, record lengthi pair per slot. The firstcomponent (record offset) is a `pointer' to the record, as shown in Figure 7.8; it is theoffset in bytes from the start of the data area on the page to the start of the record.Deletion is readily accomplished by setting the record offset to -1. Records can bemoved around on the page because the rid, which is the page number and slot number(that is, position in the directory), does not change when the record is moved; onlythe record offset stored in the slot changes.

12. RECORD FORMATS

In this section we discuss how to organize fields within a record. While choosing a wayto organize the fields of a record, we must take into account whether the fields of therecord are of fixed or variable length and consider the cost of various operations on therecord, including retrieval and modification of fields.Before discussing record formats, we note that in addition to storing individual records,information that is common to all records of a given record type (such as the numberof fields and field types) is stored in the system catalog, which can be thought of asa description of the contents of a database, maintained by the DBMS.

12.1. Fixed-Length Records

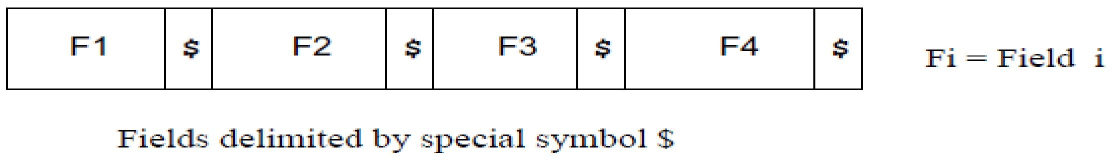
This avoids repeated storage of the same information with each record of a given type,the value in this fieldis of the same length in all records, and the number of fields is also fixed. The fieldsof such a record can be stored consecutively, and, given the address of the record, theaddress of a particular field can be calculated using information about the lengths ofpreceding fields, which is available in the system catalog. This record organization isillustrated in Figure.

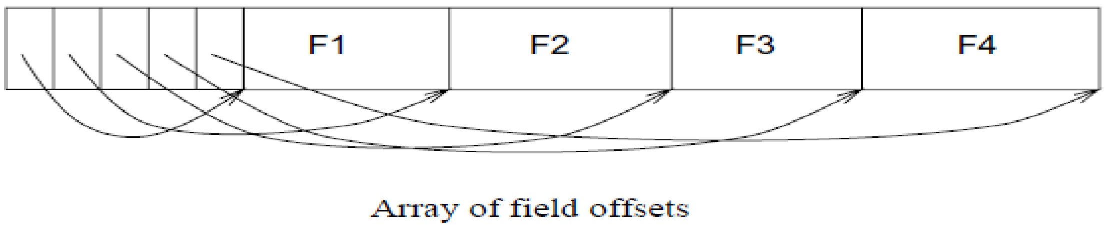


Organization of Records with fixed length fields

12.2. Variable-Length Records

In the relational model, every record in a relation contains the same number of fields.If the number of fields is fixed, a record is of variable length only because some of itsfields are of variable length.One possible organization is to store fields consecutively, separated by delimiters (whichare special characters that do not appear in the data itself). This organization requiresa scan of the record in order to locate a desired field.An alternative is to reserve some space at the beginning of a record for use as an arrayof integer offsets,the ith integer in this array is the starting address of the ithfield value relative to the start of the record. Note that we also store an offset to the end ofthe record; this offset is needed to recognize where the last field ends. Both alternativesare illustrated in Figure.





Alternative Record Organizations for variable length records

The second approach is typically superior. For the overhead of the offset array, weget direct access to any field. We also get a clean way to deal with null values. Anull value is a special value used to denote that the value for a field is unavailable orinapplicable. If a field contains a null value, the pointer to the end of the field is setto be the same as the pointer to the beginning of the field. That is, no space is usedfor representing the null value, and a comparison of the pointers to the beginning and the end of the field is used to determine that the value in the field is null.

Variable-length record formats can obviously be used to store fixed-length records aswell; sometimes, the extra overhead is justified by the added flexibility, because issuessuch as supporting null values and adding fields to a record type arise with fixed-lengthrecords as well.

Having variable-length fields in a record can raise some subtle issues, especially whena record is modified.Modifying a field may cause it to grow, which requires us to shift all subsequentfields to make space for the modification in all three record formats presentedabove.A record that is modified may no longer fit into the space remaining on its page.If so, it may have to be moved to another page.